По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 5-93286-020-0, название «Программирование на Perl, 3-е издание» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.

Programming Perl

Third Edition

Larry Wall, Tom Christiansen
& Jon Orwant

O'REILLY®

Программирование на Perl

Третье издание

Ларри Уолл, Том Кристиансен
и Джон Орвант

Санкт-Петербург
2002

Ларри Уолл, Том Кристиансен и Джон Орвант

Программирование на Perl

Перевод С. Маккавеева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научные редакторы	*К. Иванов, В. Рижий*
Редактор	*В. Овчинников*
Корректура	*С. Беляева*
Верстка	*А. Дорошенко*

Уолл Л., Кристиансен Т., Орвант Д.
Программирование на Perl. – Пер. с англ. – СПб: Символ-Плюс, 2002. – 1152 с., ил.
ISBN 5-93286-020-0

Первое издание «Программирование на Perl», ставшее непререкаемой библией языка, вышло в 1991 году. Сейчас Perl завоевал широкую популярность, что и потребовало нового, уже третьего издания, содержащего как введение в язык Perl для новичков в программировании, так и отличный справочник по языку.

Ларри Уолл – создатель Perl и один из авторов этой книги. По образованию он еще и лингвист. Возможно, поэтому Perl стал необычайно гибким языком, где одного и того же можно достичь многими путями, как это прекрасно демонстрирует автор. Уже написано много книг, в которых рассматриваются многочисленные возможности Perl, однако только в этой книге рассказывается, зачем эти возможности были созданы и как их использовать в полную силу. Особенно полезен Perl в системном администрировании и веб-программировании.

Что нового в третьем издании? Практически все. Оно не просто расширено в соответствии с релизом Perl 5.6, но и полностью реорганизовано и усилено множеством примеров. Большая часть разделов радикально переработана, например, разделы, посвященные объектно-ориентированному программированию и регулярным выражениям. Кроме того, добавлено много новых глав, рассматривающих работу с профилями, Unicode, потоки, компилирование, документацию pod и внутреннюю организацию Perl.

ISBN 5-93286-020-0
ISBN 0-596-00027-8 (англ)

© Издательство Символ-Плюс, 2002
Authorized translation of the English edition © 2000 O'Reilly & Associates Inc. This translation is published and sold by permission of O'Reilly & Associates Inc., the owner of all rights to publish and sell the same.

Все права на данное издание защищены Законодательством РФ, включая право на полное или частичное воспроизведение в любой форме. Все товарные знаки или зарегистрированные товарные знаки, упоминаемые в настоящем издании, являются собственностью соответствующих фирм.

Издательство «Символ-Плюс». 193148, Санкт-Петербург, ул. Пинегина, 4,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.
Подписано в печать 28.02.2002. Формат 70x100 1/16 . Печать офсетная.
Объем 72 печ. л. Тираж 3000 экз. Заказ N
Отпечатано с диапозитивов в Академической типографии «Наука» РАН
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Предисловие 13

В погоне за счастьем 13
Что обновилось в этом издании 18
Стандартный дистрибутив 19
Электронная документация 21
Печатная документация 25
Дополнительные источники 27
Соглашения, принятые в этой книге 28
Благодарности 29
Хотим услышать ваши отзывы 30

Часть I. Общий обзор 31

1. Обзор Perl 33

Введение 33
Естественные и искусственные языки 34
Стандартный пример 48
Дескрипторы файлов 52
Операторы 54
Управляющие структуры 61
Регулярные выражения 68
Обработка списков 74
Чего вы не знаете, то вам не навредит (сильно) 76

Часть II. Анатомия Perl 79

2. Всякая всячина 81

Атомы 82
Молекулы 83
Встроенные типы данных 85
Переменные 87
Имена 88
Скалярные значения 94
Контекст 105

Списочные значения и массивы . . . 108
Хеши . . . 113
Таблицы имен и дескрипторы файлов . . . 115
Операторы ввода . . . 116

3. Унарные и бинарные операторы . . . 123

Термы и списковые операторы (слева) . . . 126
Оператор «стрелка» . . . 127
Автоинкрементирование и автодекрементирование . . . 128
Возведение в степень . . . 129
Идеографические унарные операторы . . . 129
Операторы связывания . . . 130
Мультипликативные операторы . . . 131
Аддитивные операторы . . . 132
Операторы сдвига . . . 133
Именованные унарные операторы и операторы проверки файлов . . . 133
Операторы сравнения . . . 138
Операторы равенства . . . 138
Операторы поразрядного действия . . . 139
Логические операторы в стиле C (короткого действия) . . . 140
Оператор диапазона . . . 141
Условный оператор . . . 142
Операторы присваивания . . . 144
Оператор запятой . . . 146
Списковые операторы (справа) . . . 147
Логические and, or, not и xor . . . 147
Операторы C, отсутствующие в Perl . . . 148

4. Операторы и объявления . . . 149

Простые операторы . . . 150
Составные операторы . . . 151
Операторы if и unless . . . 153
Операторы Loop . . . 154
Голые блоки . . . 162
goto . . . 165
Глобальные объявления . . . 166
Объявления с областью видимости . . . 168
Прагмы . . . 176

5. Поиск по шаблону . . . 179

Бестиарий регулярных выражений . . . 180
Операторы поиска по шаблону . . . 183
Метасимволы и метазнаки . . . 199
Классы символов . . . 208

Квантификаторы . . . 219
Позиции . . . 222
Захват и кластеризация . . . 225
Чередование . . . 230
Управление процессом . . . 232
Сложные шаблоны . . . 247

6. Подпрограммы . . . 262

Синтаксис . . . 262
Семантика . . . 264
Передача ссылок . . . 269
Прототипы . . . 271
Атрибуты подпрограмм . . . 277

7. Форматы . . . 280

Переменные форматов . . . 284
Нижние колонтитулы . . . 286

8. Ссылки . . . 288

Что такое ссылка? . . . 289
Создание ссылок . . . 291
Использование жестких ссылок . . . 297
Символические ссылки . . . 310
Фигурные скобки, квадратные скобки и кавычки . . . 311

9. Структуры данных . . . 315

Массивы массивов . . . 316
Хеши массивов . . . 323
Массивы хешей . . . 325
Хеши хешей . . . 327
Хеши функций . . . 330
Более сложные записи . . . 331
Сохранение структур данных . . . 334

10. Пакеты . . . 335

Таблицы имен . . . 340
Автозагрузка . . . 344

11. Модули . . . 346

Использование модулей . . . 346
Создание модулей . . . 349
Замещение встроенных функций . . . 353

12. Объекты 355

Краткая памятка по объектно-ориентированному жаргону 355
Система объектов Perl 357
Вызов методов 358
Создание объектов 364
Наследование классов 369
Деструкторы экземпляров 378
Управление данными экземпляров 380
Управление данными класса 391
Резюме 394

13. Перегрузка 395

Прагма overload 396
Обработчики перегрузки 397
Перегружаемые операторы 398
Конструктор копий (=) 405
Когда обработчик перегрузки отсутствует (nomethod и fallback) 406
Константы перегрузки 407
Открытые функции перегрузки 409
Наследование и перегрузка 409
Перегрузка на этапе исполнения 410
Диагностика перегрузки 410

14. Связанные переменные 411

Связывание скаляров 413
Связывание массивов 421
Связывание хешей 427
Связывание указателей файлов 433
Тонкая ловушка при отвязывании 444
Модули для связывания в CPAN 447

Часть III. Perl как технология 449

15. Unicode 451

Байты и символы 453
Действие символьной семантики 455
Осторожно, работают 人 459

16. Межпроцессное взаимодействие 461

Сигналы 462
Файлы 469
Каналы 477
System V IPC 485
Сокеты 489

17. Потоки 497

Модель процессов 498
Модель потоков 499

18. Компиляция 515

Жизненный цикл программ на Perl 516
Компилирование кода 518
Выполнение кода 524
Серверы компиляторов 527
Генераторы кода 528
Средства разработки кода 530
Компилятор и интерпретатор: авангардизм и ретро 532

19. Интерфейс командной строки 537

Обработка команд 537
Переменные окружения 554

20. Отладчик Perl 557

Использование отладчика 558
Команды отладчика 560
Настройка отладчика 569
Автоматическое выполнение 573
Поддержка отладчика 575
Профайлер Perl 578

21. Внутри и снаружи 582

Как работает Perl 583
Внутренние типы данных 583
Расширение Perl (использование C из Perl) 584
Вызов Perl из C 591
Мораль «сей басни» 596

Часть IV. Perl как культура 597

22. CPAN 599

Каталог modules архива CPAN 601
Использование модулей CPAN 604
Создание модулей CPAN 606

23. Защита данных 609

Обработка ненадежных данных 610
Обработка ошибок синхронизации 622
Работа с ненадежным кодом 629

24. Распространенные приемы программирования 639

Обычные промахи новичков 639
Эффективность 648
Стиль программирования 658
Свободный разговор на Perl 662
Генерирование программ 672

25. Переносимость программ Perl 677

Перевод строки 678
Остроконечники, тупоконечники и ширина чисел 680
Файлы и файловые системы 681
Взаимодействие с системой 682
Межпроцессное взаимодействие (IPC) 683
Внешние подпрограммы (XS) 683
Стандартные модули 684
Дата и время 684
Многоязычность 685
Стиль 685

26. Документация в формате POD 686

О pod в двух словах 686
Трансляторы и модули Pod 695
Создание собственных инструментов для работы с pod 697
Ловушки pod 700
Документирование программ Perl 701

27. Культура Perl 703

История развития 703
Поэзия Perl 706

Часть V. Справочный материал 709

28. Специальные имена 711

Специальные имена, сгруппированные по типам 712
Специальные переменные в алфавитном порядке 714

29. Функции 736

Функции Perl по категориям 739
Функции Perl в алфавитном порядке 741

30. Стандартная библиотека Perl 895

Библиотековедение 895
Обзор библиотеки Perl 897

31. Модули прагм 900

use attributes 901
use autouse 902
use base 903
use blib 904
use bytes 904
use charnames 905
use constant 906
use diagnostics 908
use fields 910
use filetest 912
use integer 913
use less 914
use lib 914
use locale 916
use open 917
use overload 917
use re 918
use sigtrap 919
use strict 922
use subs 925
use vars 925
use warnings 926

32. Стандартные модули 930

Перечень по типу 931
Benchmark 941
Carp 943
CGI 944
CGI::Carp 944
Class::Struct 945
Config 946
CPAN 947
Cwd 947
Data::Dumper 948
DB_File 948
Dumpvalue 950
English 950
Errno 951
Exporter 951
Fatal 952
Fcntl 953
File::Basename 953
File::Compare 954
File::Copy 955
File::Find 955

File::Glob 956
File::Spec 959
File::stat 960
File::Temp 960
FileHandle 961
Getopt::Long 964
Getopt::Std 965
IO::Socket 966
IPC::Open2 967
IPC::Open3 967
Math::BigInt 968
Math::Complex 969
Math::Trig 969
Net::hostent 969
POSIX 970
Safe 972
Socket 973
Symbol 974
Sys::Hostname 975
Sys::Syslog 976
Term::Cap 977
Text::Wrap 978
Time::Local 978
Time::localtime 979
User::grent 980
User::pwent 980

33. Диагностические сообщения 982

Глоссарий 1045

Алфавитный указатель 1084

Предисловие

В погоне за счастьем

Perl – язык, с помощью которого вы сделаете свою работу.

Конечно, если эта работа – программирование, то теоретически ее можно сделать с помощью любого «полного» компьютерного языка. Но опыт показывает, что компьютерные языки различаются не столько *возможностью* что-либо сделать, сколько *легкостью*, с которой это достигается. На одном полюсе находятся так называемые языки четвертого поколения, с помощью которых можно легко делать одни вещи и почти невозможно другие. На другом полюсе – так называемые языки с промышленными возможностями (industrial-strength languages), посредством которых одинаково трудно делать почти все.

Perl не таков. Если сказать кратко, то он разработан так, чтобы легко решать простые задачи, сохраняя возможность решать трудные.

Что это за «простые задачи», которые должны решаться легко? Разумеется, те, которые мы решаем изо дня в день. Нам нужен язык, с помощью которого легко работать с числами и текстом, файлами и каталогами, компьютерами и сетями, а в особенности – с программами. Он должен позволять легко запускать внешние программы и просматривать результаты их работы в поиске интересных данных. Он должен позволять легко отправлять эти интересные данные другим программам, способным обрабатывать их особым образом. Он должен также позволять нам легко разрабатывать собственные программы, изменять их и производить отладку. И, конечно, наши собственные программы должны легко компилироваться и запускаться, а также быть переносимыми на любую современную операционную систему.

Все это, а также многое другое, делает Perl.

Первоначально разработанный как интегрирующий язык для Unix, Perl давно распространился на большинство других операционных систем. Поскольку Perl выполняется почти везде, он является одной из наиболее переносимых сред программирования, имеющихся на сегодняшний день. Чтобы написать переносимые программы на C или C++, необходимо расставить все эти странные пометки `#ifdef` для каждой операционной системы. Для обеспечения переносимости программ на Java нужно разбираться в индивидуальных особенностях всех новых реализаций. Для создания переносимых

сценариев командного процессора нужно помнить синтаксис всех команд каждой версии операционной системы и пытаться найти общий знаменатель, благодаря которому они, как можно надеяться, будут работать всюду. А чтобы создавать переносимые программы на Visual Basic потребуется дать более гибкое определение понятию «переносимость». :-)

В Perl удается избежать таких проблем, сохраняя при этом многие преимущества других языков и добавляя собственные чудеса. Чудеса связаны с рядом причин: практичностью набора его функций, изобретательностью сообщества разработчиков Perl и богатством, которое предоставляет движение open source в целом. Однако в значительной мере чудеса обусловлены гибридной природой Perl. У Perl смешанное происхождение, и многообразие средств всегда рассматривалось в нем как сила, а не слабость. Perl – это язык, говорящий: «Дайте мне вашу усталость, вашу бедность»[1]. Если вы чувствуете себя словно стиснутым в толпе и стремитесь «дышать свободой», то Perl – для вас.

Perl охватывает различные культуры. Его взрывное распространение в значительной мере питалось стремлением бывших системных Unix-программистов взять с собой как можно больше из «старого мира». Для них Perl является квинтэссенцией культуры Unix, оазисом в пустыне «невозможности перейти из одного места в другое». Существует, однако, и движение в обратном направлении: веб-дизайнеры, работающие под Windows с удовольствием обнаруживают возможность запустить свои Perl-программы на Unix-сервере своей компании без их дополнительной переработки.

Хотя Perl особенно популярен среди системных программистов и разработчиков для Интернета, это связано лишь с тем, что они первыми его открыли; аудитория Perl значительно шире. Получивший при создании скромный статус языка обработки текста, Perl превратился в сложный язык программирования общего назначения с богатой средой разработки программ, укомплектованной отладчиками, профайлерами, формировщиками перекрестных ссылок, компиляторами, библиотеками, синтаксически-ориентированными редакторами и всеми остальными атрибутами «настоящего» языка программирования – если они вам требуются. Но все они относятся к поддержке возможностей решения сложных задач, с чем справляются многие другие языки. Уникальность Perl в том, что он никогда не терял из виду возможность легко решать задачи простые.

Поскольку Perl является одновременно мощным и доступным средством, он постоянно используется во всех мыслимых сферах – от аэрокосмической техники до молекулярной биологии, от математики до лингвистики, от графики до обработки документов, от обработки баз данных до сетевого администрирования. Perl используется теми, кому необходимо быстро проанализировать или преобразовать большие объемы данных, будь то последова-

---

[1] «Give me your tired, your poor, Your huddled masses yearning to breathe free...» – слова, которыми начинается надпись на основании Статуи Свободы. – *Примеч. ред.*

тельность генов ДНК, веб-страницы или фьючерсы на свиные внутренности. В самом деле в сообществе Perl бытует шутка, что очередной крупный кризис на рынке ценных бумаг может разразиться из-за ошибки, сделанной кем-нибудь в своем сценарии Perl. (Светлой стороной окажется то, что безработные рыночные аналитики окажутся, так сказать, востребованными.)

Существует много слагаемых успеха. Perl как проект open source достиг его еще до того, как это движение получило свое название. Perl свободно распространяется, и так будет всегда. Каждый может работать с Perl так, как сочтет удобным, и на основе очень либеральной политики лицензирования. Если вы занимаетесь коммерческой деятельностью и хотите воспользоваться Perl, можете смело идти вперед. Разрешается применять Perl в разрабатываемых коммерческих приложениях бесплатно и без ограничений. А для тех, у кого возникнет проблема, которую сообщество Perl не сможет решить, существует последняя инстанция: сам исходный код. Сообщество Perl не занимается продажей своих профессиональных тайн под видом «обновлений». Сообщество Perl никогда не «выйдет из дела» и не оставит вас с брошенным на произвол судьбы продуктом.

Безусловно, популярности Perl способствует его бесплатное распространение. Но этого недостаточно для объяснения феномена Perl, поскольку многие бесплатно распространяемые пакеты не преуспели. Дело не в том, что он бесплатен; он доставляет удовольствие. Люди чувствуют желание творить на Perl, поскольку он дает свободу самовыражения: можно выбирать между целями оптимизации – скоростью работы компьютера или скоростью программирования, между многословием и выразительностью, между «читабельностью» и возможностью поддержки или повторного использования, или переносимости, или обучаемости, или поучительности. Можно оптимизировать даже непонятность, если принять участие в конкурсе на самую непонятную программу – Obfuscated Perl Contest.

Perl может предоставить все эти степени свободы, поскольку это язык с раздвоением личности. Это одновременно очень простой язык и очень богатый язык. Perl собрал отовсюду хорошие идеи и поместил их в простую в использовании логическую структуру. Для тех, кому он просто нравится, Perl – это *Practical Extraction and Report Language* (практический язык извлечения данных и создания отчетов). Для тех, кто любит его, Perl – это *Pathologically Eclectic Rubbish Lister* (паталогически эклектичный язык для распечатки чепухи). А для многочисленных минималистов Perl кажется проявлением бесцельной избыточности. Но это хорошо. Редукционисты должны существовать (в основном среди физиков). Редукционисты стремятся разъять целое на части. Мы, все остальные, пытаемся собрать их вместе.

Во многих отношениях Perl is a simple language (простой язык). Не требуется знать множества особых заклинаний, чтобы скомпилировать программу на Perl – ее можно просто выполнить как пакетный файл или сценарий оболочки. Типы и структуры Perl просты в использовании и понимании. Perl не налагает свои произвольные ограничения на данные – строки и массивы мо-

гут быть сколь угодно велики, лишь бы хватило оперативной памяти, и их организация позволяет им легко увеличиваться по мере надобности. Perl не требует изучения новых синтаксиса и семантики, в значительной мере заимствуя их из других языков, с которыми вы можете быть знакомы (например, C, *awk*, BASIC, Python, английский и греческий). На практике почти любой программист может прочесть хорошо написанный код Perl и составить себе представление о том, что он делает.

Очень важно, что нет необходимости изучить Perl полностью, чтобы начать писать полезные программы. Можно начать изучение Perl с «тонкого конца». Вы можете программировать на языке «детский лепет Perl», и мы обещаем не смеяться над этим. Точнее, мы обещаем смеяться не более, чем над первыми творческими попытками ребенка. Многие идеи Perl заимствованы из естественного языка, и одна из лучших состоит в том, что можно использовать лишь подмножество языка, если его достаточно для того, чтобы передать мысль. В культуре Perl приемлема любая степень владения языком. Полицию по охране языка мы к вам не пришлем. Сценарий Perl будет «правильным», если выполнит задачу прежде, чем начальник вас уволит.

Будучи во многих отношениях простым, Perl является и богатым языком, который нужно долго изучать. Это расплата за возможность решать сложные задачи. Хотя понадобится некоторое время на освоение всех средств Perl, вы будете рады иметь в своем распоряжении расширенные возможности, когда настанет момент и они вам понадобятся.

Благодаря своему происхождению Perl был богатым языком даже тогда, когда служил «просто» языком преобразования данных, предназначенным для перемещения по файлам, просмотра больших объемов текста, создания и получения динамических данных и вывода легко форматируемых отчетов, основанных на этих данных. Но в какой-то момент начался расцвет Perl. Он стал также и языком для работы с файловой системой, управления процессами, администрирования баз данных, программирования в архитектуре клиент-сервер, создания безопасных программ, управления данными в Сети и даже для объектно-ориентированного и функционального программирования. Эти возможности не были просто механически присоединены к Perl – каждая новая синергически работает с остальными, поскольку с самого начала Perl проектировался как интегрирующий язык.

Но Perl может объединять в единое целое не только свои собственные функции. Он разработан как модульно расширяемый язык. Perl позволяет быстро разрабатывать, программировать, отлаживать и разворачивать приложения, а также при необходимости без труда расширять функциональные возможности этих приложений. Perl можно встраивать в другие языки, а также встраивать другие языки в Perl. С помощью механизма импорта модулей можно использовать эти внешние определения, как если бы они были встроенными функциями Perl. Объектно-ориентированные внешние библиотеки сохраняют свою объектную ориентированность в Perl.

Perl полезен и в других отношениях. В отличие от строго интерпретируемых языков, таких как командные файлы и сценарии оболочки, которые одно-

временно компилируют и выполняют одну команду, Perl сначала быстро компилирует программу в промежуточный формат. Подобно любому другому компилятору, он осуществляет различного вида оптимизации и мгновенно реагирует на любые ошибки – от синтаксических и семантических до неудачи при связывании с библиотеками. Когда компилирующий интерфейс Perl оказывается удовлетворен вашей программой, он передает промежуточный код на выполнение интерпретатору (либо какому-либо из нескольких модулей серверов, способных создавать текст на C в виде байт-кодов). Все это выглядит сложным, однако компилятор и интерпретатор работают весьма эффективно, и обычный цикл компиляции-прогона-исправления занимает считанные секунды. В совокупности с большим количеством предоставляемых Perl функций амортизации отказов эта возможность короткого времени оборота делает Perl языком, на котором действительно возможно быстрое прототипирование. Позже, по мере совершенствования программы вы можете повысить требовательность к себе и заставить себя программировать, применяя меньше искусства, основанного на интуиции, и больше дисциплины. Perl и в этом окажет содействие, если его хорошо об этом попросить.

Perl также способствует созданию более защищенных программ. Помимо всех обычных интерфейсов защиты, предоставляемых другими языками, Perl защищает от случайных ошибок в системе безопасности посредством уникального механизма трассировки данных, автоматически определяющего данные, которые поступили из ненадежного источника, и предотвращает выполнение опасных операций. Наконец, Perl позволяет создавать специальные защищенные отсеки, в которых можно безопасно выполнять код сомнительного происхождения с запрещением опасных операций.

Парадоксально, но самая большая помощь, которую Perl может оказать программисту, связана не столько с самим Perl, сколько с людьми, которые с ним работают. Сообщество Perl составляют люди, которые более чем кто-либо другой готовы прийти на помощь. Если считать, что в движении Perl есть что-то благочестивое, то оно лежит в самой его сердцевине. Ларри хотел, чтобы сообщество было чем-то вроде рая, и в целом его желание пока осуществляется. Внесите и свой вклад в то, чтобы оно таким и оставалось.

Возможно, вы изучаете Perl, поскольку хотите спасти мир либо просто из любопытства, либо вам приказал делать это ваш начальник – в любом случае этот учебник позволит познакомиться как с основами, так и со сложными вопросами. И хотя мы не намереваемся учить вас программированию, проницательный читатель что-то приобретет как от искусства, так и от науки программирования. Мы подстрекаем вас развивать в себе три великие добродетели программиста: *лень*, *нетерпение* и *самомнение* (*laziness*, *impatience*, *hubris*). Мы надеемся, что, читая эту книгу, вы найдете ее местами довольно занимательной (и местами очень занимательной). Если этого окажется недостаточно для того, чтобы вы не заснули, то постоянно напоминайте себе, что изучение Perl повысит ценность вашего резюме. Так что читайте дальше.

Что обновилось в этом издании

Пожалуй, почти все.

Даже там, где мы сохранили удачные места из предыдущего издания (а полагаем, что таких было достаточно много), мы существенно пересмотрели и переработали материал, стремясь достичь нескольких целей. Во-первых, мы хотели повысить доступность книги для тех, кто не получил подготовки в области вычислительной техники. Мы уменьшили предполагаемый объем знаний читателя. В то же время мы старались оживить изложение в надежде, что те, кто уже частично знаком с обсуждаемым предметом, не заснут во время чтения.

Во-вторых, мы хотели представить последние достижения в развитии самого Perl. С этой целью мы не стеснялись сообщать о текущем состоянии разработок, даже сознавая, что они все еще находятся в экспериментальной стадии. Несмотря на то что ядро Perl стоит как скала в течение ряда лет, скорость разработки некоторых его экспериментальных расширений подчас весьма впечатляет. Мы честно сообщаем о тех случаях, когда, по нашему мнению, электронная документация может оказаться более надежной, чем сведения в нашей книге. Perl – это язык рабочего человека, и мы не боимся называть вещи своими именами.

В-третьих, мы хотели облегчить поиск нужного материала, поэтому уменьшили объем глав, сделали их более вразумительными, а также объединили главы в части по смыслу. Вот структура нового издания:

Часть 1 «Общий обзор»

Начать всегда труднее всего. В этой части базовые идеи Perl излагаются в неформальном виде – устройтесь поудобнее в вашем любимом кресле. Не будучи полным учебным руководством, эта часть просто предлагает быстрое начало, которое может потребоваться не каждому читателю. Поищите в разделе «Печатная документация» книги, которые могут лучше соответствовать вашему стилю учебы.

Часть 2 «Анатомия Perl»

В этой части проводится глубокое и ничем не сдерживаемое обсуждение внутреннего устройства языка на всех уровнях абстракции – от типов данных, переменных и регулярных выражений до подпрограмм, модулей и объектов. Читатель получит хорошее представление о том, как работает язык, а также несколько советов по правильному проектированию программ. (А тех, кто никогда не использовал язык с поиском по шаблону, ждет особое удовольствие.)

Часть 3 «Perl как технология»

Многое можно делать с помощью одного только Perl, но в этой части вы достигнете более высокого уровня мастерства. Узнаете о том, как заставить Perl пройти через все препятствия, которые поставит перед ним ваш компьютер, – от обработки, взаимодействия процессов и многопоточнос-

ти до компилирования, вызова, отладки и профилирования, а также создания собственных внешних расширений на C или C++ или интерфейсов к имеющимся API. Perl будет счастлив побеседовать с любым интерфейсом на вашем компьютере да, пожалуй, и любом другом компьютере в Интернете, если позволят погодные условия.

Часть 4 «Perl как культура»

Каждому ясно, что у культуры должен быть свой язык, но сообществу Perl всегда было ясно, что у языка должна быть культура. В этой части мы рассматриваем программирование на Perl как человеческую деятельность, являющуюся частью реального мира людей. Мы даем также много советов относительно того, как можно заниматься самосовершенствованием и как сделать, чтобы ваши программы приносили больше пользы людям.

Часть 5 Справочный материал

Здесь мы собрали главы, в которых читатель сможет найти что-либо в алфавитном порядке – от специальных переменных и функций до стандартных модулей и прагм. Глоссарий окажется особенно полезен тем, кто не знаком с жаргоном, используемым в вычислительной технике. Например, те, кто не знает, что такое «прагма», могут прямо сейчас посмотреть значение этого слова. (А тем, кто не знает значение слова «такое», мы не можем помочь ничем.)

Стандартный дистрибутив

В настоящее время большинство поставщиков операционных систем включают Perl в качестве стандартной составляющей своей системы. На момент написания данной книги Perl входит в стандартные дистрибутивы AIX, BeOS, BSDI, Debian, DG/UX, DYNIX/ptx, FreeBSD, IRIX, LynxOS, Mac OS X, OpenBSD, RedHat, SINIX, Slackware, Solaris, SuSE и Tru64. Некоторые компании поставляют Perl на отдельных CD с бесплатным программным обеспечением или через группы обслуживания клиентов. Сторонние поставщики, такие как ActiveState, предоставляют откомпилированные дистрибутивы для ряда операционных систем, в том числе производимых Microsoft.

Даже если поставщик включил Perl в стандартный дистрибутив, в конечном итоге, возможно, понадобится откомпилировать и установить Perl самостоятельно. В результате вы будете знать, что ваша версия является самой свежей, и сможете сами выбрать, куда установить библиотеки и документацию. Также можно будет решить, следует ли скомпилировать Perl с поддержкой дополнительных расширений, таких как многопоточность, большие файлы или множество низкоуровневых опций отладки, доступ к которым осуществляется через ключ командной строки `<option>-D</option>`. (Отладчик уровня пользователя поддерживается всегда.)

Проще всего загрузить комплект исходного кода Perl, указав броузеру домашнюю страницу на *www.perl.com*, где находятся также ссылки на компи-

лированные двоичные модули для платформ, компиляторы C для которых затерялись.

Можно также направиться прямо в CPAN (Comprehensive Perl Archive Network, архив Perl, описанный в главе 22 «CPAN») по адресу *http://www.perl.com/CPAN* или *http://www.cpan.org*. Если работа с ними окажется слишком медленной (а это может случиться, поскольку они очень популярны), следует найти зеркальный сервер CPAN поблизости от себя. Ниже приведены URL лишь некоторых из зеркал, разбросанных по всему свету и числом превышающих сотню:

http://www.funet.fi/pub/languages/perl/CPAN
ftp://ftp.funet.fi/pub/languages/perl/CPAN/
ftp://ftp.cs.colorado.edu/pub/perl/CPAN/
ftp://ftp.cise.ufl.edu/pub/perl/CPAN/
ftp://ftp.perl.org/pub/perl/CPAN/
http://www.perl.com/CPAN-local
http://www.cpan.org/
http://www.perl.org/CPAN/
http://www.cs.uu.nl/mirror/CPAN/
http://CPAN.pacific.net.hk/

Первая парочка из этого списка на сайте *funet.fi* указывает на основное хранилище CPAN. Список всех остальных сайтов CPAN находится в файле *MIRRORED.BY*, поэтому лучше сначала скачать этот файл, а затем выбрать понравившееся зеркало. Доступ к одним осуществляется через FTP, к другим – через HTTP (если вы спрятались за корпоративным брандмауэром, это может иметь значение). Редиректор *http://www.perl.com/CPAN* пытается сделать этот выбор вместо вас. При желании можно изменить свой выбор позже.

Получив исходный код и распаковав его в каталог, нужно прочесть файлы *README* и *INSTALL*, чтобы узнать, как скомпилировать и скомпоновать Perl. Может иметься и файл *INSTALL.*`platform`, где `platform` представляет платформу вашей операционной системы.

Если данная платформа является разновидностью Unix, то команды, необходимые для получения, конфигурирования, сборки и установки Perl, могут быть примерно следующие. Во-первых, необходимо выбрать команду, с помощью которой будет получен исходный код. Можно воспользоваться *ftp* :

```
% ftp ftp://ftp.funet.fi/pub/languages/perl/CPAN/src/latest.tar.gz
```

(Не бойтесь заменить адрес на ближайшее зеркало CPAN. Конечно, если вы живете в Финляндии, то это и есть ваше ближайшее зеркало CPAN.) При невозможности воспользоваться *ftp* можно осуществить загрузку через Интернет с помощью броузера или средства командной строки:

```
% wget http://www.funet.fi/pub/languages/perl/CPAN/src/latest.tar.gz
```

Теперь нужно распаковать, сконфигурировать, собрать и установить:

```
% tar zxf latest.tar.gz          Или сначала gunzip, tar xf.
% cd perl-5.6.0                  Или 5.* для других версий.
% sh Configure -des              Принимает ответы по умолчанию.
% make test && make install      Установка обычно производится
                                  суперпользователем.
```

При этом используется обычная среда разработки на C, так что если компилятора C на машине нет, скомпилировать Perl не удастся. Посмотрите в каталоге CPAN *ports* самые свежие данные по каждой платформе относительно поставки Perl вместе с дистрибутивом (и если так, то какой версии), возможности обойтись стандартным комплектом исходного кода или необходимости специального переноса. Ссылки для загрузки даются для тех систем, которые обычно требуют специальных переносов, или систем от поставщиков, для которых отсутствие компилятора C – нормальная практика (правильнее было бы сказать, «ненормальная» практика).

Электронная документация

Обширная электронная документация по Perl входит в состав его стандартного дистрибутива. (О бумажной документации говорится в следующем разделе.) Дополнительная документация появляется, как только устанавливается новый модуль из CPAN.

Упоминая в этой книге «страницу руководства Perl», мы имеем в виду комплект электронных страниц руководства по Perl, который находится на вашем компьютере. Под *страницей электронного руководства* (*manpage*) будем понимать просто файл с документацией, для чтения которого не обязательно иметь Unix-программу *man*. Страницы руководства Perl могут быть установлены даже как страницы HTML, особенно на системах, отличных от Unix.

Электронные страницы руководства по Perl разделены на несколько секций, поэтому можно легко найти нужное, не продираясь через сотни страниц текста. Поскольку страница верхнего уровня называется просто *perl*, то в Unix команда *man perl* должна привести именно на нее.[1] Эта страница, в свою очередь, ведет на более специальные страницы. Например, *man perlre* выведет страницу руководства по регулярным выражениям Perl. Команда *perldoc* часто работает в тех системах, в которых не работает команда *man*. На компьютерах Macintosh следует выполнить команду *Shuck*. Конкретный перенос может также предоставлять страницы руководства по Perl в формате HTML или родном для системы формате подсказки. Уточните этот вопрос у своего системного администратора – если, конечно, сами не являетесь им.

---

[1] Если при этом открывается нечто необозримое, то, вероятно, вы обращаетесь к древнему руководству версии 4. Проверьте, не указывает ли MANPATH на места, где можно производить археологические раскопки. (Введите *perldoc perl*, чтобы узнать, как настроить MANPATH соответственно выдаче команды *perl –V:man.dir*.)

Навигация по стандартным страницам руководства

В незапамятные времена (если говорить о Perl, то имеется в виду 1987 год) страница руководства *perl* была кратким документом объемом около 24 печатных страниц. Например, раздел по регулярным выражениям занимал всего два абзаца. (Этого было достаточно при условии знакомства с *egrep*.) В некоторых отношениях с тех пор изменилось почти все. Если считать стандартную документацию, различные утилиты, сведения о переносах на различные платформы и множество стандартных модулей, то наберется свыше 1500 печатных страниц документации, разбросанных по многим отдельным страницам электронного руководства. (И это без учета модулей из CPAN, которые могут быть у вас установлены, и в немалом количестве.)

Но в других отношениях не изменилось ничего: по-прежнему жива страница руководства *perl* и по-прежнему это лучшее место для начала, когда не известно, откуда начать. Разница в том, что, попав туда, нельзя остановиться. Документация – это больше не кустарное производство, это торговый пассаж с сотнями магазинов. Войдя в дверь, необходимо найти плакат, который поможет определить, где находится лавка или универсальный магазин, продающие то, зачем вы пришли. Конечно, освоившись в торговом центре, как правило, заранее знаешь, куда направиться.

Вот некоторые вывески:

Страница руководства	Что освещает
perl	Какие страницы руководства по Perl имеются
perldata	Типы данных
perlsyn	Синтаксис
perlop	Операторы и их приоритеты
perlre	Регулярные выражения
perlvar	Предопределенные переменные
perlsub	Подпрограммы
perlfunc	Встроенные функции
perlmod	Как использовать модули Perl
perlref	Ссылки
perlobj	Объекты
perlipc	Межпроцессное взаимодействие
perlrun	Как выполнять команды Perl плюс ключи
perldebug	Отладка
perldiag	Диагностические сообщения

Это лишь небольшой отрывок, но он состоит из важных частей. Как видно, если нужны сведения об операторе, то для этого подойдет *perlop*. А если нуж-

но что-то выяснить о предопределенных переменных, следует искать в *perlvar*. Если получено непонятное диагностическое сообщение, идите на *perldiag*. И так далее.

В стандартное руководство по Perl входит список часто задаваемых вопросов (FAQ). Он разбит на следующие девять разных страниц:

Страница руководства	Что освещает
perlfaq1	Общие вопросы по Perl
perlfaq2	Получение Perl и сведения о нем
perlfaq3	Инструменты программирования
perlfaq4	Обработка данных
perlfaq5	Файлы и форматы
perlfaq6	Регулярные выражения
perlfaq7	Общие вопросы языка Perl
perlfaq8	Взаимодействие с системой
perlfaq9	Сетевое взаимодействие

Некоторые страницы руководства содержат замечания по специфике платформ:

Страница руководства	Что освещает
perlamiga	Перенос на Amiga
perlcygwin	Перенос на Cygwin
perldos	Перенос на MS-DOS
perlhpux	Перенос на HP-UX
perlmachten	Перенос на Power MachTen
perlos2	Перенос на OS/2
perlos390	Перенос на OS/390
perlvms	Перенос на DEC VMS
perlwin32	Перенос на MS-Windows

(Относительно переносов см. также главу 25 «Переносимость программ Perl» и каталог CPAN *ports*, описанный выше.)

Поиск на страницах электронного руководства

Нет необходимости читать все 1500 печатных страниц, чтобы найти иголку в стоге сена. Старая поговорка гласит, что нельзя «грепнуть» (*grep*) мертвое дерево.[1] Кроме обычных средств поиска, присутствующих в большинстве

[1] Не забудьте, что при необходимости можно заглянуть в глоссарий.

программ просмотра документов, в версии 5.6.1 каждая основная страница руководства имеет собственные средства поиска и отображения. Можно осуществлять поиск в отдельных страницах, используя имя страницы руководства в качестве команды и передавая регулярное выражение Perl (см. главу 5 «Поиск по шаблону») в качестве шаблона для поиска:

```
% perlop comma

% perlfunc split

% perlvar ARGV

% perldiag 'assigned to typeglob'
```

Если не известно точно, в каком месте документации находятся требующиеся сведения, можно расширить поиск. Например, для поиска во всех FAQ воспользуйтесь командой *perlfaq* (которая тоже является страницей руководства):

```
% perlfaq round
```

Команда *perltoc* (которая тоже является страницей руководства) осуществляет поиск в общем для всех страниц руководства оглавлении:

```
% perltoc typeglob
perl5005delta: Undefined value assigned to typeglob
perldata: Typeglobs and Filehandles
perldiag: Undefined value assigned to typeglob
```

Можно также проводить поиск во всем электронном руководстве по Perl, включая заголовки, описания и примеры, используя команду *perlhelp*:

```
% perlhelp CORE::GLOBAL
```

Подробности см. в странице руководства по *perldoc*.

Страницы руководства, не принадлежащие Perl

Когда мы ссылаемся на документацию, не принадлежащую Perl, как в *getitimer*(2), ссылка указывает на страницу руководства *getitimer* из раздела 2 *Unix Programmer's Manual*.[1] Страницы руководства для системных вызовов, таких как *getitimer*, могут отсутствовать на системах, отличных от Unix, но это может быть и правильно, потому что системные вызовы на них все равно нельзя использовать. Для тех, кому действительно требуется документация

[1] В разделе 2 должны содержаться данные только о прямых вызовах операционной системы. (Они часто называются системными обращениями («system calls»), но мы неуклонно называем их в этой книге системными вызовами (*syscalls*), чтобы не спутать с функцией system, не имеющей отношения к системным вызовам. Однако между системами есть некоторые различия в том, какие вызовы реализованы как системные, а какие – как обращения к библиотекам C, поэтому есть вероятность, что *getitimer*(2) будет обнаружена в разделе 3.

по команде, системному вызову или библиотечной функции Unix, скажем, что многие организации поместили свои страницы руководства в Интернете. Задав поиск «+crypt(3)+manual» на AltaVista, можно найти много адресов.

Хотя страницы руководства по Perl верхнего уровня обычно устанавливаются в секции 1 стандартных каталогов *man*, мы не будем добавлять (1) к названиям таких страниц руководства в нашей книге. Их легко узнать, поскольку они имеют вид «`perl`*бубубу*».

Печатная документация

Тем, кто хочет больше узнать про Perl, рекомендуем некоторые издания:

- *Perl 5 Pocket Reference*, 3d ed., by Johan Vromans, O'Reilly, 2000 (Карманный справочник по Perl, 3-е изд., Йохан Вроманс). Эта маленькая брошюра служит удобным справочником по Perl.
- *Perl Cookbook*, by Tom Christiansen and Nathan Torkington, O'Reilly, 1998 («Perl: библиотека программиста», Т. Кристиансен, Н. Торкингтон, изд-во «Питер», 2000). Эта книга дополняет ту, которую вы сейчас держите в руках.
- *Elements of Programming with Perl*, by Andrew L. Johnson Manning, 1999 (Программирование на Perl, Эндрью Л. Джонсон). Эта книга предназначена для обучения с азов непрограммистов тому, как нужно программировать вообще и делать это с помощью Perl в частности.
- *Learning Perl*, 2d ed., by Randal Schwartz and Tom Christiansen, O'Reilly, 1997 («Изучаем Perl», Р. Шварц, Т. Кристиансен, BHV-Киев). Эта книга научит системных администраторов и программистов для Unix 30 основным процентам Perl, которые им понадобятся в 70 процентах случаев. Эрик Олсон (Erik Olson) переработал эту книгу, создав вариант для программистов на системах Microsoft под названием *Learning Perl for Win32 Systems*.
- *Perl: The Programmer's Companion*, by Nigel Chapman, Wiley, 1997 (Perl: спутник программиста, Найджел Чапмен). Эта чудесная книга предназначена для профессионалов в вычислительной технике и программировании безотносительно к используемой платформе. Perl освещается кратко, но полно.
- *Mastering Regular Expressions*, by Jeffrey Friedl, O'Reilly, 1997 («Регулярные выражения. Библиотека программиста», Д. Фридл, изд-во «Питер», 2001). Хотя в книге не освещены последние нововведения в регулярных выражениях Perl, она является ценным справочником для всех, кто хочет знать, как в действительности работают регулярные выражения.
- *Object Oriented Perl*, by Damian Conway, Manning, 1999 (Объектно-ориентированный Perl, Дамиан Конвей). Для начинающих и опытных разработчиков объектно-ориентированных программ. Эта удивительная книга

излагает обычные и тайные приемы создания мощных объектных систем на Perl.

- *Mastering Algorithms with Perl*, by Jon Orwant, Jarkko Hietaniemi, and John Macdonald, O'Reilly, 1999 (Perl: алгоритмы, Джон Орвант, Йаркко Хьетаниеми и Джон Макдональд). Все полезные приемы из курса вычислительных алгоритмов, но без тягостных доказательств. Книга освещает фундаментальные и полезные алгоритмы, относящиеся к графам, текстам, множествам и ряду других областей.
- *Writing Apache Modules with Perl and C*, by Lincoln Stein and Doug MacEachern, O'Reilly, 1999 (Разработка модулей Apache на Perl и C, Л. Штайн). Это руководство по веб-программированию учит тому, как расширить возможности веб-сервера, особенно с использованием модуля `mod_perl` для создания быстро выполняющихся сценариев CGI и доступа из Perl к Apache API.
- *The Perl Journal*, редактируемый Джоном Орвантом (Jon Orwant). Этот ежеквартальный журнал, издаваемый программистами и для программистов, регулярно публикует глубокие материалы по программированию, технические приемы, последние новости и прочее.

Существует много других книг и публикаций по Perl, и по старческой дряхлости мы наверняка позабыли упомянуть некоторые хорошие. (Из милосердия мы пренебрегли обязанностью отметить некоторые плохие издания.)

Помимо перечисленных публикаций, относящихся к Perl, мы рекомендуем следующие книги. Они не посвящены непосредственно Perl, но их удобно иметь под рукой для получения справки, совета или вдохновения.

- *The Art of Computer Programming*, by Donald Knuth, vol. 1, *Fundamental Algorithms*; vol. 2, *Seminumerical Algorithms*; vol. 3, *Sorting and Searching*, Addison-Wesley, 1998.[1]
- *Introduction to Algorithms*, by Cormen, Leiserson, and Rivest MIT Press and McGraw-Hill, 1990 (Введение в алгоритмы, Кормен, Лейзерсон и Райвест).
- *Algorithms in C: Fundamental Data Structures, Sorting, Searching*, 3d ed., by Robert Sedgewick, Addison-Wesley, 1997 («Фундаментальные алгоритмы на C++. Анализ, структуры данных, сортировка, поиск», 3-е издание, Р. Седжвик, изд-во «Диасофт»).
- *The Elements of Programming Style*, by Kernighan and Plauger, Prentice-Hall, 1988 (Элементы стиля программирования, Керниган и Плаугер).
- *The Unix Programming Environment*, by Kernighan and Pike, Prentice-Hall, 1984 (Среда разработки Unix, Керниган и Пайк).

---

[1] Д. Кнут «Искусство программирования», том 1 «Основные алгоритмы»; том 2 «Получисленные алгоритмы»; том 3 «Сортировка и поиск»; 3-е издание, изд-во «Вильямс», 2000.

- *POSIX Programmer's Guide*, by Donald Lewine, O'Reilly, 1991 (Posix: руководство программиста, Д. Левин).
- *Advanced Programming in the UNIX Environment*, by W. Richard Stevens, Addison-Wesley, 1992 (Профессиональное программирование в среде Unix, У. Сивенс).
- *TCP/IP Illustrated*, vols. 1–3, by W. Richard Stevens, Addison-Wesley, 1994–1996 (Иллюстрированный TCP/IP, У. Стивенс).
- *The Lord of the Rings*, by J. R. R. Tolkien (последнее издание: Houghton Mifflin, 1999).

Дополнительные источники

Интернет – чудесное изобретение, и все мы продолжаем открывать возможности его наиболее полного использования. (Конечно, некоторые предпочитают «открывать» Интернет так, как Толкиен открывал Центр Земли.)

Perl в Сети

Посетите домашнюю страницу Perl *http://www.perl.com/*. На ней рассказывается, что нового есть в мире Perl, содержатся исходные коды и переносы, тематические статьи, документация, расписания конференций и многое другое.

Посетите также веб-страницу Perl Mongers *http://www.perl.org*, чтобы взглянуть на Perl с точки зрения широких масс, так сказать, на «корни», которые бурно разрастаются во всех частях света, за исключением Южного полюса, где их приходится держать в закрытом помещении. Местные группы PM проводят регулярные небольшие встречи, на которых можно обменяться профессиональным опытом с другими хакерами Perl, живущими в той же части света.

Телеконференции (группы новостей Usenet)

Телеконференции Perl служат огромным, хотя подчас беспорядочным, источником сведений о Perl. Начать можно с *comp.lang.perl.moderated* – модерируемой телеконференции с невысоким потоком сообщений, содержащей объявления и технические дискуссии. Благодаря модерированию телеконференция вполне читаема.

Группа *comp.lang.perl.misc* с интенсивным потоком сообщений обсуждает все – от технических вопросов до философии Perl, игр Perl и поэзии Perl. Как и сам Perl, *comp.lang.perl.misc* предназначена для пользы, и в ней можно задать любой глупый вопрос.[1]

---

[1] Конечно, некоторые вопросы такие глупые, что на них и отвечать не стоит (особенно если на них уже есть ответы в страницах руководства и FAQ. Зачем просить помощи в телеконференции, если можно самому найти ответ, потратив меньше времени, чем занимает ввод запроса?)

В группе *comp.lang.perl.tk* обсуждается, как применять в Perl популярный комплект Tk. Группа *comp.lang.perl.modules* обсуждает разработку и использование модулей Perl, которые служат лучшим способом создания повторно используемого кода. Когда вы будете читать эту книгу, могут появиться новые телеконференции *comp.lang.perl.нечто*, поэтому поищите их.

Если для доступа в Usenet используется не обычная программа чтения новостей, а веб-броузер, введите `"news:"` и имя телеконференции, чтобы получить доступ к одной из указанных здесь телеконференций. (Это возможно, только если у вас есть доступ к серверу новостей.) В другом варианте, если для поиска в Usenet используются такие службы, как AltaVista или Deja, укажите `"*perl*"` в качестве телеконференций, которые нужно найти.

Другой телеконференцией, материалы которой могут понадобиться, по крайней мере, тем, кто занимается CGI-программированием для Интернета, является *comp.infosystems.www.authoring.cgi*. Хотя, строго говоря, это не группа Perl, большинство обсуждаемых в ней программ написаны на Perl. К ней полезно обратиться по вопросам Perl, связанным с Сетью, если только вы не используете `mod_perl` под Apache; тогда можно подписаться на *comp.infosystems.www.servers.unix*.

Сообщения об ошибках

В том маловероятном случае, если ошибка обнаружена в самом Perl, а не в вашей программе, нужно постараться воспроизвести ее в минимальном по объему контрольном примере и сообщить о ней с помощью программы *perlbug*, поставляемой с Perl. Дополнительные сведения можно найти на *http://bugs.perl.org*.

Соглашения, принятые в этой книге

Некоторые из принятых нами соглашений более подробно обсуждаются в соответствующих местах. Соглашения по коду обсуждаются в разделе «Стиль программирования» главы 24 «Распространенные приемы программирования». В некотором смысле наши лексические соглашения представлены в глоссарии (наш словарь).

В книге приняты следующие типографские соглашения:

Курсив

Предназначен для URL, страниц руководства, маршрутов и программ. Новые термины тоже выделяются курсивом при первом появлении в тексте. Для многих из этих терминов можно найти альтернативные определения в глоссарии, если недостаточно тех, которые имеются в тексте.

`Моноширинный`

Используется в примерах и в обычном тексте для вывода кода. Данные обозначаются моноширинным шрифтом и заключаются в кавычки (""), не являющиеся частью значения.

Моноширинный полужирный

Моноширинный полужирный шрифт применяется для выделения ключей командной строки. Это позволяет различать, скажем, ключ вывода предупредительных сообщений **-w** и оператор проверки файла -w. Этот шрифт используется также в примерах для обозначения текста, вводимого буквально.

Моноширинный курсив

Служит для обозначения общих элементов кода, вместо которых необходимо подставить конкретные значения.

Мы приводим массу примеров, большинство из которых представляют собой фрагменты более крупных программ. Некоторые примеры являются завершенными программами, что можно увидеть по начальной строке #!. Почти все наши более длинные программы начинаются со строки:

```
#!/usr/bin/perl
```

В других примерах приводится текст, который следует ввести в командной строке. Мы используем % для обозначения приглашения оболочки:

```
% perl -e 'print "Hello, world.\n"'
Hello, world.
```

Этот стиль – типичный образец стандартной командной строки Unix, в которой одиночные кавычки представляют «наиболее закавыченный» формат. На других системах соглашения по использованию кавычек и символов заполнителей могут быть иными. Например, многие интерпретаторы команд в MS-DOS и VMS требуют двойные, а не одиночные кавычки при задании аргументов, содержащих пробелы и символы-заполнители.

Благодарности

Здесь мы публично приносим благодарность нашим рецензентам, что должно компенсировать все грубости, сказанные им в частном порядке. Это: Тод Миллер (Todd Miller), Шэрон Хопкинс Рауэзан (Sharon Hopkins Rauenzahn), Рич Рауэнзан (Rich Rauenzahn), Пол Маркес (Paul Marquess), Пол Грасси (Paul Grassie), Натан Торкингтон (Nathan Torkington), Йохан Вроманс (Johan Vromans), Джефф Хемер (Jeff Haemer), Гурусами Сарати (Gurusamy Sarathy), Глория Уолл (Gloria Wall), Дэн Сугальский (Dan Sugalski) и Эбигайл (Abigail).

Хотим выразить особую благодарность Тиму О'Рейли (Tim O'Reilly) и его Associates за побуждение авторов к написанию книг, которые читателям приятно читать.

Хотим услышать ваши отзывы

Мы протестировали и выверили все сведения, представленные в этой книге с возможно большей тщательностью, но читатели могут обнаружить, что некоторые функции претерпели изменения (или даже что мы допустили ошибки!). Просим сообщить обо всех найденных ошибках, а также пожеланиях для последующих изданий по адресу:

O'Reilly & Associates, Inc.

101 Morris Street Sebastopol, CA 95472

1-800-998-9938 (в США и Канаде)

1-707-829-0515 (международный/местный)

1-707-829-0104 (факс)

Можно также посылать сообщения электронной почтой. Чтобы быть включенным в наш лист почтовой рассылки или запросить каталог, посылайте письма по адресу *info@oreilly.com*. Чтобы задать технический вопрос или сообщить свои комментарии по поводу этой книги, шлите почту по адресу *bookquestions@oreilly.com*.

Для этой книги существует сайт, на котором мы публикуем список замеченных ошибок и прочие сведения, относящиеся к верблюду:

http://www.oreilly.com/catalog/pperl3

Там вы найдете также тексты всех примеров из этой книги, которые можно загрузить, чтобы не вводить все самим, как это пришлось делать нам.

I

Общий обзор

1

Обзор Perl

Введение

Мы полагаем, что Perl является языком, который легко изучить и использовать, и надеемся убедить вас в своей правоте. Одним из обстоятельств, делающих Perl простым, является то, что не нужно произносить много предварительных слов, прежде чем сказать то, что вы собираетесь сказать. Во многих языках программирования необходимо объявить типы, переменные и подпрограммы, которые предполагается использовать, прежде чем написать хотя бы одну строчку исполняемого кода. И это правильно, что необходимы объявления, если речь идет о сложных задачах, требующих сложных структур данных. Но для многих более простых, повседневных задач хотелось бы иметь язык программирования, на котором можно просто сказать:

```
print "Howdy, world!\n";
```

и программа это сделает.

Таким языком и является Perl. На практике этот пример представляет собой законченную программу[1], и если послать ее интерпретатору Perl, тот выведет на экран `"Howdy, world"`. (`\n` в этом примере обеспечивает перевод строки в конце вывода.)

---

[1] Или «сценарий» (script), «приложение» (application), «исполняемый объект» (executable), «штуковина» (doohickey). Кому что больше нравится.

Вот и все. *После* того как вы сказали то, что хотели, тоже не требуется произносить много слов. В отличие от многих языков программирования, Perl считает, что «вывалиться» из программы – нормальный способ ее завершения. Конечно, при желании вы *можете* явно вызвать функцию `exit`, как *можете* и объявить некоторые переменные, либо даже сделать *обязательным* объявление всех переменных. Но право выбора принадлежит вам. В Perl можно поступать «так, как должно», как бы вы это ни определили.

Есть много других причин, по которым Perl легко использовать, но перечислять сейчас их все бессмысленно, поскольку этому и посвящена наша книга. Говорят, что дьявол проявляет себя в частностях, но Perl старается облегчить вашу жизнь и в трудных ситуациях. На любом уровне Perl заботится о том, чтобы доставить вас туда, куда нужно, с минимумом суеты и максимумом удовольствия. Вот почему многие программисты на Perl ходят с глупыми ухмылками на физиономиях.

Эта глава представляет собой обзор, поэтому мы не пытаемся представить Perl с рациональной стороны. Равно как не претендуем на полноту изложения или логичность. Для этого предназначены следующие главы. Вулканическим натурам, андроидам и другим аналогично настроенным лицам следует пропустить этот обзор и перейти сразу к главе 2 «Всякая всячина», где плотность информации выше. С другой стороны, тем, кому нужен учебник с тщательно выстроенным продвижением вперед, возможно, следует обратиться к прекрасной книге Рэндала Шварца (Randal Schwartz) «Learning Perl» (опубликованную O'Reilly & Associates). Но и данную книгу пока не выбрасывайте.

Эта глава представляет Perl *другому* полушарию вашего мозга, тому, которое называют ассоциативным, артистическим, управляющим эмоциями или просто впитывающим. Для этого Perl рассматривается с разных точек зрения, чтобы дать читателю о нем такое же ясное представление, какое слепцы могли получить о слоне (ощупывая его). Возможно, нам удастся больше: мы все-таки имеем дело с верблюдом (см. обложку). Надеемся, что хотя бы одно из этих представлений Perl поможет вам меньше горбатиться.

Естественные и искусственные языки

Языки были изобретены людьми и для блага людей. В анналах компьютерных наук этот факт случайно оказался позабытым.[1] Поскольку (допуская вольность речи) можно сказать, что создателем Perl оказался лингвист, предполагалось, что этот язык будет работать так же гладко, как естественные языки. На деле это охватывает множество аспектов, поскольку естественные языки хорошо работают одновременно на нескольких уровнях. Можно было бы перечислить многие из этих лингвистических принципов, но самым важным принципом архитектуры языка является тот, что прос-

[1] Точнее, об этом факте вспоминают от случая к случаю.

тые вещи должны делаться просто, а сложные вещи должны быть реализуемы. (На самом деле, здесь два принципа.) Они могут показаться вам очевидными, однако многие языки программирования не удовлетворяют хотя бы одному из этих условий.

Естественные языки удовлетворяют обоим условиям, поскольку люди непрерывно пытаются выражать как простые, так и сложные вещи, и поэтому язык развивается так, чтобы справляться с обеими задачами. Возможность развития была заложена в Perl на этапе его конструирования, и он действительно эволюционировал со времени своего создания. Вклад в эволюцию Perl вносили многие люди на протяжении ряда лет. Мы часто шутим, что верблюд – это лошадь, которую проектировала комиссия, но если вдуматься, то верблюд достаточно хорошо приспособлен для жизни в пустыне. Эволюция привела верблюда к относительной самодостаточности. (С другой стороны, в результате эволюции верблюд не стал лучше пахнуть. Как и Perl.) Это одна из многих странных причин, по которым мы сделали верблюда талисманом для Perl, но к лингвистике это имеет отдаленное отношение.

Когда кто-то произносит слово «лингвистика», у многих возникает одно из двух представлений: либо они начинают думать о словах, либо начинают думать о предложениях. Но слова и предложения являются лишь двумя удобными способами членения речи. Те и другие можно разбить на более мелкие смысловые единицы или объединить в более крупные. А смысл каждой единицы существенно зависит от синтаксического, семантического и практического контекста, в котором находится эта единица. В естественном языке есть слова разных видов: существительные, глаголы и т. д. Если произнести слово «dog» (собака) изолированно, в отрыве от контекста, представляется существительное, хотя можно использовать это слово и в другом смысле. Это значит, что существительное может выполнять функции глагола, прилагательного или наречия, в зависимости от контекста. If you dog a dog during the dog days of summer, you'll be a dog tired dogcatcher.[1]

Perl также по-разному оценивает слова в зависимости от контекста. Далее мы увидим, как он это делает. Помните просто, что Perl пытается понять, о чем идет речь, как это делает любой хороший слушатель. Perl усиленно старается соблюсти со своей стороны условия сделки. Скажите просто, чего вы хотите, и Perl обычно «сообразит». (Если только вы не говорите чепухи – анализатор Perl понимает Perl значительно лучше, чем английский или суахили.)

Но вернемся к существительным. Существительное может быть именем конкретного объекта или родового класса объектов без указания того, на который из них производится ссылка в данный момент. Это различие проводится в большинстве языков программирования, когда конкретная величина называется значением, а общая – переменной. Значение где-то существует,

---

[1] Пример многозначности dog и зависимости от контекста. – *Примеч. перев.*

Возможно, читатель устал, как собака, от всей этой лингвистической трескотни. Но мы хотим, чтобы было понятно, чем Perl отличается от обычных компьютерных языков!

а вот переменная в течение срока своей жизни связывается с одним или более значениями. Поэтому, кто бы ни осуществлял интерпретацию, переменная должна отслеживать эту связь. Интерпретатор может находиться в мозгу человека или в компьютере.

Синтаксис переменных

Переменная является просто удобным способом хранения чего-либо, именованным местонахождением, в котором программист может найти нечто требующееся ему, когда оно позднее понадобится. Как и в реальной жизни, есть разного типа места для хранения, одни из которых достаточно скрыты, а другие открыты для всех. Некоторые хранилища временные, а другие – более постоянные. Ученые в области вычислительной техники любят говорить об «области видимости» переменных, подразумевая под этим выражением именно то, что в нем сказано. В Perl имеются достаточно удобные средства решения проблем области видимости, с которыми читатель в свое время будет иметь удовольствие познакомиться. Оно еще не наступило. (Те, кто испытывает любопытство, могут поискать прилагательные `local`, `my` и `our` в главе 29 «Функции», либо заглянуть в раздел «Объявления, ограниченные областью видимости» главы 4 «Операторы и объявления».)

В данный момент полезнее провести классификацию переменных по типу данных, которые они могут содержать. Как и в английском языке, в Perl основное различие проводится между данными в единственном и во множественном числе. Строки и числа являются данными в единственном числе, а списки строк или чисел – во множественном. (А когда мы доберемся до объектно-ориентированного программирования, вы обнаружите, что типичный объект выглядит единичным снаружи, но множественным изнутри – как группа студентов.) Переменную в единственном числе мы называем *скаляром*, а во множественном – *массивом*. Поскольку строка может храниться в скалярной переменной, можно написать такую несколько более длинную (и снабженную комментариями) версию нашего первого примера:

```
$phrase = "Howdy, world!\n";          # Присвоить значение переменной.
print $phrase;                        # Вывести переменную
```

Обратите внимание, что нам не пришлось предварительно определять, какой тип имеет переменная `$phrase`. Символ `$` указывает Perl на то, что `phrase` представляет собой скалярную переменную, т. е. содержит единственное значение. Напротив, массив должен начинаться с символа `@`. (Запоминание облегчается тем, что `$` является стилизованным «s», от «scalar», тогда как `@` – это стилизованное «a», от «array».)

В Perl имеются некоторые другие типы переменных с такими непривлекательными именами, как «хеш» (hash), «дескриптор» (handle) и «запись таблицы имен» (typeglob). Подобно скалярам и массивам, переменным этих типов также предшествуют разыменовывающие префиксы (funny characters – забавные символы). Вот полный список разыменовывающих префиксов, которые могут вам встретиться:

Тип	Символ	Пример	Что обозначает
Скаляр	`$`	`$cents`	Отдельное значение (число или строку)
Массив	`@`	`@large`	Список значений с числовым ключом
Хеш	`%`	`%interest`	Группа значений, ключом доступа к которым служит строка
Подпрограмма	`&`	`&how`	Фрагмент кода Perl, который может быть вызван
Typeglob	`*`	`*struck`	Все, что имеет имя `struck`

Некоторые пуристы языка указывают на эти разыменовывающие префиксы как на причину отвращения к Perl. Это поверхностный взгляд. У этих символов много достоинств, не последним из которых является возможность подстановки, или интерполяции (interpolation), этих символов в строки без всякого дополнительного синтаксиса. Кроме того, сценарии Perl легко читать (тем, кто взял на себя труд изучить Perl!), поскольку существительные отличаются от глаголов. В язык могут добавляться новые глаголы без ущерба для старых сценариев. (Мы уже говорили, что в Perl на этапе проектирования была заложена возможность развития.) И аналогия с существительными приведена не зря: есть масса примеров в английском и других языках, когда требуются грамматические маркеры существительных. Вот так мы думаем! (Мы думаем.)

Единственное число

В приведенном выше примере мы видели, что скаляру можно присваивать новое значение с помощью оператора = подобно тому, как это делается во многих других языках программирования. Скалярным переменным можно присваивать скалярные значения любого вида: целые числа, числа с плавающей запятой, строки и даже такие таинственные вещи, как ссылки на другие переменные или объекты. Есть много способов создания этих значений, которые должны присваиваться.

Как и в оболочке Unix[1], можно использовать различные механизмы расстановки кавычек для создания значений разного вида. Двойные кавычки (double quotes) производят *интерполяцию переменных*[2] и *интерполяцию обратной косой черты* (например, превращение `\n` в символ перевода строки), тогда как одиночные кавычки подавляют интерполяцию. А обратные

[1] Здесь и всюду, говоря «Unix», мы подразумеваем любую операционную систему, похожую на Unix, в том числе BSD, Linux и, конечно, Unix.

[2] Иногда называемую программистами для оболочки «подстановкой», но мы предпочитаем сохранить это слово для использования в Perl в других целях. Поэтому, пожалуйста, называйте это интерполяцией. Мы используем этот термин в текстологическом смысле («этот отрывок является гностической интерполяцией»), а не в математическом («данная точка на графике получена интерполяцией между двумя другими точками»).

кавычки (наклоненные влево) вызывают выполнение внешней программы и возврат ее выдачи, которая перехватывается как одна строка, содержащая все строки выдачи.

```
$answer = 42;                # целое число
$pi = 3.14159265;            # "действительное" число
$avocados = 6.02e23;         # научное обозначение
$pet = "Camel";              # строка
$sign = "I love my $pet";    # строка с интерполяцией
$cost = 'It costs $100';     # строка без интерполяции
$thence = $whence;           # значение другой переменной
$salsa = $moles * $avocados; # гастрохимическое выражение
$exit = system("vi $file");  # числовой статус выполнения команды
$cwd = `pwd`;                # строка выдачи команды
```

И хотя мы еще не рассказали о необычных значениях, следует указать, что скаляры могут содержать ссылки на другие структуры данных, в том числе подпрограммы и объекты.

```
$ary = \@myarray;            # ссылка на именованный массив
$hsh = \%myhash;             # ссылка на именованный хеш
$sub = \&mysub;              # ссылка на именованную подпрограмму

$ary = [1,2,3,4,5];          # ссылка на анонимный массив
$hsh = {Na => 19, Cl => 35}; # ссылка на анонимный хеш
$sub = sub { print $state }; # ссылка на анонимную подпрограмму

$fido = new Camel "Amelia";  # ссылка на объект
```

Если используется переменная, которой еще ни разу не присваивалось значение, то неинициализированная переменная автоматически начинает существовать, когда в ней возникает необходимость. В соответствии с принципом минимизации неожиданности переменная создается с нулевым значением: `""` или `0`. В зависимости от контекста использования, переменные автоматически интерпретируются как строки, числа или значения «true» (истина) или «false» (ложь), обычно называемые булевыми значениями. Вспомните, как важен контекст в естественном языке. В Perl разные операторы предполагают получение в качестве параметров различного вида одиночных значений, и мы будем говорить, что данные операторы предоставляют для этих параметров скалярный контекст. Иногда мы будем более конкретно говорить, что оператор предоставляет числовой, строковый или булев контекст для этих параметров. (Ниже мы расскажем о списочном контексте, который противопоставляется скалярному.) Perl автоматически преобразует данные к виду, требуемому текущим контекстом, в соответствии со здравым смыслом. Предположим, например, что вы сказали следующее:

```
$camels = '123';
print $camels + 1, "\n";
```

Первоначальным значением `$camels` является строка, но она преобразуется в число, чтобы прибавить к ней единицу, а затем обратно в строку, чтобы вы-

вести ее как 124. Символ перевода строки, представленный "\n", тоже находится в строковом контексте, но поскольку он изначально является строкой, преобразование не требуется. Обратите, однако, внимание на двойные кавычки: использование одиночных кавычек, т. е. '\n', привело бы к появлению строки из двух символов, а именно косой черты и «n», которая никак не может представлять перевод строки.

Итак, в некотором смысле двойные и одиночные кавычки предоставляют еще один способ задания контекста. Интерпретация содержимого закавыченной строки зависит от того, какого вида кавычки используются. (Далее мы рассмотрим некоторые другие операторы, синтаксически работающие как кавычки, но использующие строку некоторым особым способом, например, для поиска по шаблону или подстановки. Все они тоже действуют как строки в двойных кавычках. Контекст *двойных кавычек* является в Perl «интерполирующим» и предоставляется многими операторами, которые не напоминают двойные кавычки.)

Аналогично ссылка ведет себя как ссылка в «разыменовывающем» контексте и как обычная скалярная величина в других случаях. Например, можно сказать так:

```
$fido = new Camel "Amelia";
if (not $fido) { die "dead camel"; }
$fido->saddle();
```

Здесь создается ссылка на объект Camel, которая помещается в переменную $fido. В следующей строке мы проверяем значение $fido как булевой величины на равенство «true» и возбуждаем исключительную ситуацию (т. е. жалуемся, что она не истинна); в данном случае это означает, что конструктор new Camel не смог создать требуемый объект Camel. Однако в последней строке мы обращаемся с $fido как со ссылкой, запрашивая метод saddle() для объекта, содержащегося в $fido и представляющего собой Camel, поэтому Perl ищет метод saddle() для объектов Camel. Подробнее об этом сказано ниже. Сейчас просто запомните, что в Perl важен контекст, поскольку благодаря ему Perl узнает, что вам требуется, без ваших явных указаний, необходимых во многих других языках программирования.

Множественное число

В некоторых типах переменных содержатся множественные значения, связанные вместе логически. В Perl есть два типа многозначных переменных: массивы и хеши. Во многих случаях они ведут себя как скаляры: они начинают существовать, когда в них возникает надобность, ничего не содержа в себе. Но в отличие от скаляров, при присвоении им значений они предоставляют *списочный* контекст в правой части оператора присваивания, а не скалярный.

Массивы и хеши также различаются между собой. Массив следует использовать, когда нужно найти что-либо по номеру. Хеш применяется для того, чтобы найти что-либо по имени. Эти два понятия дополняют друг друга.

Часто можно наблюдать использование массива для перевода номера месяца в название и соответствующего хеша для перевода названия месяца обратно в номер. (Однако хеши могут содержать не только числа. К примеру, можно создать хеш, который будет переводить названия месяцев в названия камней, соответствующих месяцам.)

Массивы. *Массив* – это упорядоченный список скаляров, к которым обращаются[1] по их местоположению в списке. Список может содержать числа, строки или смесь того и другого. (Он может также содержать ссылки на вложенные массивы (субмассивы) и вложенные хеши (субхеши).) Чтобы присвоить массиву значение в виде списка, нужно просто объединить вместе значения с помощью круглых скобок:

```
@home = ("кушетка", "стул", "стол", "печка");
```

И наоборот, используя `@home` в списочном контексте, например, с правой стороны присвоения списку, мы получим на выходе тот самый список, который ввели. Например, присвоить значения, взятые из массива, четырем скалярным переменным можно так:

```
($potato, $lift, $tennis, $pipe) = @home;
```

Это называется присвоением списка. Логически оно происходит параллельно, поэтому можно обменять значениями две переменные с помощью:

```
($alpha,$omega) = ($omega,$alpha);
```

Как и в языке C, нумерация массивов начинается с нуля (zero-based), поэтому, работая с элементами массива с первого по четвертый, следует обращаться к ним по индексам от 0 до 3.[2] Индексы массивов заключаются в квадратные скобки [вот такие], поэтому при выборе отдельного элемента массива нужно ссылаться на него так – `$home[n]`, где `n` является значением индекса (на единицу меньшим номера элемента), который нам нужен. Посмотрите на следующий пример. Поскольку мы имеем дело со скалярным элементом, ему обязательно должен предшествовать символ `$`.

Если вы хотите присваивать значения элементам массива по одному, предшествующее присвоение можно записать так:

```
$home[0] = "кушетка";
$home[1] = "стул";
$home[2] = "стол";
$home[3] = "печка";
```

---

[1] Или ключом (индексом, нижним индексом) для доступа к которым является местоположение. Выберите то, что вам нравится.

[2] Если это кажется вам непривычным, считайте индекс смещением, т. е. числом предшествующих элементов массива. Очевидно, перед первым элементом нет других элементов, поэтому его смещение равно 0. Так уж думают компьютеры. (Мы так думаем.)

Поскольку массивы упорядочены, над ними можно выполнять полезные действия, такие как операции со стеком push и pop. Стек, в сущности, является просто упорядоченным списком с началом и концом. С концом в особенности. Perl рассматривает конец массива как вершину стека. (Хотя большинство программистов на Perl представляют себе массивы горизонтальными, с вершиной стека, находящейся справа.)

Хеши. *Хеш* представляет собой неупорядоченный набор скаляров, к которым обращаются[1] по некоторому строковому значению, связанному с каждым скаляром. По этой причине хеши часто называют *ассоциативными массивами*. Но это слишком длинное название для ввода с клавиатуры, а говорится о них так часто, что мы решили дать им какое-нибудь краткое и броское название. Другой причиной, по которой мы выбрали название «хеш», является желание подчеркнуть тот факт, что они неупорядочены. (Так совпало, что в их внутренней реализации используется поиск по хеш-таблице, в результате чего работа с ними происходит столь быстро вне зависимости от числа помещенных в них значений.) Однако применить к хешу push или pop нельзя: это не имеет смысла. У хеша нет начала и конца. Тем не менее, хеши являются очень мощным и полезным средством. Пока вы не начали думать на языке хешей, вы, в сущности, не начали думать на Perl. На рис. 1.1 показаны упорядоченные элементы массива и неупорядоченные (но поименованные) элементы хеша.

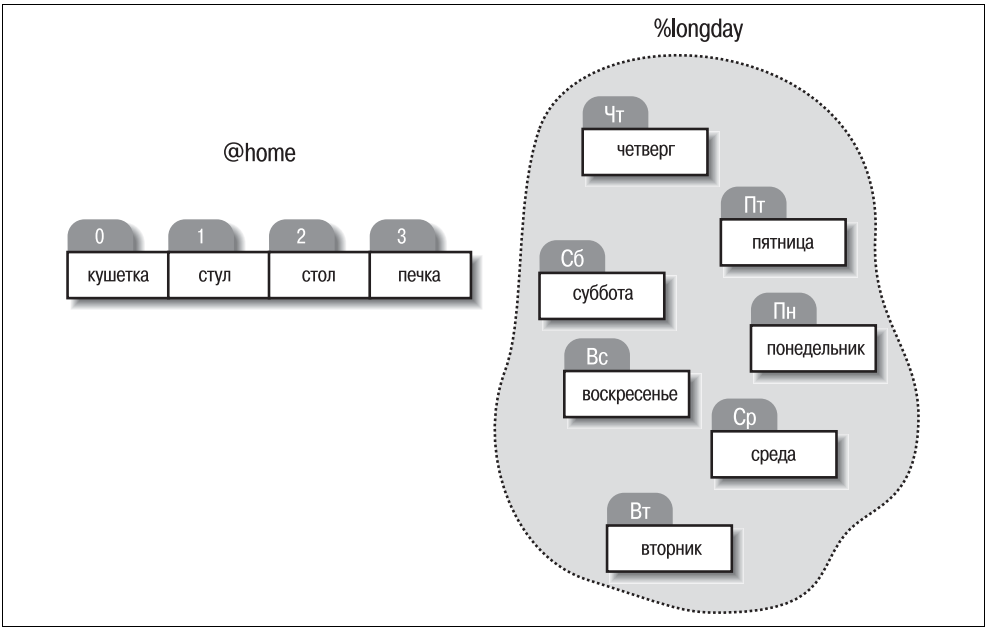


Рис. 1.1. Массив и хеш

[1] Или ключом (индексом, нижним индексом) для доступа к которым является местоположение. Выберите то, что вам нравится.

Поскольку ключи для хеша не задаются автоматически положением, при заполнении хеша необходимо задавать как значение, так и ключ. Можно присваивать хешу список, так же как и массиву, но при этом каждая *пара* элементов списка будет интерпретироваться как ключ и значение. Поскольку мы работаем с парами элементов, для пометки имен хешей используется разыменовывающий префикс %. (Если внимательно посмотреть на символ %, можно заметить ключ и значение, разделенные косой чертой. Особенно посмотрев искоса.)

Предположим, вы хотите перевести сокращенные названия дней в соответствующие полные. Можно записать такое присвоение списка:

```
%longday = ("Вс", "Воскресенье", "Пн", "Понедельник", "Вт", "Вторник",
            "Ср", "Среда", "Чт", "Четверг", "Пт",
            "Пятница", "Сб", "Суббота");
```

Но это довольно трудно читается, поэтому Perl предоставляет последовательность => (знак «равно», знак «больше») в качестве разделителя, альтернативного запятой. С использованием этой синтаксической поблажки (и творческого форматирования) становится значительно легче различить, какие строки являются ключами, а какие – связанными с ними значениями:

```
%longday = (
    "Вс" => "Воскресенье",
    "Пн" => "Понедельник",
    "Вт" => "Вторник",
    "Ср" => "Среда",
    "Чт" => "Четверг",
    "Пт" => "Пятница",
    "Сб" => "Суббота",
);
```

Можно не только присваивать список хешу, как мы сделали выше, но, если хеш упоминается в списочном контексте, последний преобразует хеш обратно в список пар «ключ/значение», правда, в причудливом порядке. Иногда это оказывается полезным. Чаще требуется извлечь список, состоящий только из ключей, для чего применяется функция, уместно названная keys. Список ключей тоже неупорядочен, но при желании его легко можно отсортировать с помощью столь же удачно названной функции sort. После этого упорядоченные ключи можно использовать для извлечения соответствующих значений в требуемом порядке.

Поскольку хеш – довольно причудливая разновидность массива, выбор отдельного элемента хеша осуществляется путем помещения ключа в фигурные скобки (своеобразные скобки, которые известны как «завитушки»[1]). Поэтому, если, скажем, требуется найти в вышеприведенном хеше значе-

[1] В английском языке, конечно, в котором для обозначения фигурных скобок существуют слова brace(s), curly brace(s)/curly brackets или просто curlies (завитушки, кудряшки). – *Примеч. ред.*

ние, ассоциированное со средой (Ср), следует использовать $longday{"Ср"}. Снова обратите внимание на то, что мы имеем дело со скалярной величиной, поэтому идентификатор начинается символом $, а не %, что указывало бы на весь хеш целиком.

Лингвистически, закодированная в хеше связь аналогична родительному падежу или притяжательному местоимению, как слово «of» в английском языке или как «s». Женой Адама является Ева, поэтому мы пишем:

```
$wife{"Адам"} = "Ева";
```

Сложности

Массивы и хеши являют собой прелестные, простые, плоские структуры данных. К несчастью, окружающий мир не всегда идет навстречу нашим попыткам чрезмерного упрощения. Иногда приходится создавать не столь прелестные, простые и плоские структуры данных. Perl позволяет осуществлять это, «делая вид», что сложные величины на самом деле являются простыми. Или другими словами, Perl позволяет манипулировать простыми скалярными ссылками, которые указывают на сложные массивы и хеши. В естественном языке мы занимаемся этим постоянно, когда используем простое существительное в единственном числе, например, «правительство» для обозначения объекта, который совершенно «закручен» и непостижим. Помимо всего прочего.

Расширим наш предыдущий пример, предположив, что вместо жены Адама мы хотим поговорить о жене Иакова. Известно, что у Иакова было четыре жены. (Проверять на себе это не нужно.) Пытаясь представить это на Perl, мы оказываемся в необычном положении, в котором хотим изобразить, что четыре жены Иакова в действительности являются одной женой. (Это тоже не надо проверять на себе.) Читатель может решить, что допустима такая запись:

```
$wife{"Иаков"} = ("Лия", "Рахиль", "Валла", Зелфа");        # НЕВЕРНО
```

Но в результате мы не получим того, что требуется, потому что в Perl даже скобки и запятые бессильны превратить список в скаляр. (Скобки предназначены для синтаксического объединения, а запятые – для синтаксического разделения.) Вместо этого необходимо явно сообщить Perl, что список требуется представить в виде скаляра. Оказывается, что достаточной силой для этого обладают квадратные скобки:

```
$wife{"Иаков"} = ["Лия", "Рахиль", "Валла", Зелфа"];        # ok
```

Эта команда создает анонимный массив и помещает ссылку на него в элемент хеша $wife{"Иаков"}. В результате получаем именованный хеш, содержащий анонимный массив. Таким способом Perl работает с многомерными массивами и вложенными структурами данных. Как и в обычных массивах и хешах, можно присваивать значения отдельным элементам наподобие следующего:

```
$wife{"Иаков"}[0] = "Лия";
$wife{"Иаков"}[1] = "Рахиль";
$wife{"Иаков"}[2] = "Валла";
$wife{"Иаков"}[3] = "Зелфа";
```

Как видите, это похоже на многомерный массив, одним индексом которого является строка, а другим – число. Чтобы показать нечто более похожее по структуре на дерево, типа вложенных структур данных, предположим, что мы хотим перечислить не только жен Иакова, но и всех сыновей каждой из его жен. В этом случае надо обращаться с хешем как со скаляром. Для этого можно использовать скобки. (Внутри каждого значения хеша мы, как и раньше, будем применять для представления массивов квадратные скобки. Но теперь у нас массив, содержащийся в хеше, содержащемся в хеше.)

```
$kids_of_wife{"Иаков"} = {
    "Лия"    => ["Рувим", "Симеон", "Левий", "Иуда", "Иссахар", "Завулон"],
    "Рахиль" => ["Иосиф", "Вениамин"],
    "Валла"  => ["Дан", "Неффалим"],
    "Зелфа"  => ["Гад", "Асир"],
};
```

Это более или менее эквивалентно следующему:

```
$kids_of_wife{"Иаков"}{"Лия"}[0]    = "Рувим";
$kids_of_wife{"Иаков"}{"Лия"}[1]    = "Симеон";
$kids_of_wife{"Иаков"}{"Лия"}[2]    = "Левий";
$kids_of_wife{"Иаков"}{"Лия"}[3]    = "Иуда";
$kids_of_wife{"Иаков"}{"Лия"}[4]    = "Иссахар";
$kids_of_wife{"Иаков"}{"Лия"}[5]    = "Завулон";
$kids_of_wife{"Иаков"}{"Рахиль"}[0] = "Иосиф";
$kids_of_wife{"Иаков"}{"Рахиль"}[1] = "Вениамин;
$kids_of_wife{"Иаков"}{"Валла"}[0]  = "Дан";
$kids_of_wife{"Иаков"}{"Валла"}[1]  = "Неффалим";
$kids_of_wife{"Иаков"}{"Зелфа"}[0]  = "Гад";
$kids_of_wife{"Иаков"}{"Зелфа"}[1]  = "Асир";
```

Как видите, добавление нового уровня во вложенную структуру данных похоже на добавление еще одного измерения в многомерный массив. Perl не препятствует программисту представлять это себе тем или иным образом, при этом внутреннее представление одинаково.

Важным моментом здесь является то, что Perl допускает представление сложной структуры данных в виде простого скаляра. На этом простом типе инкапсуляции основывается вся объектно-ориентированная структура Perl. Когда выше мы вызывали конструктор для `Camel` в виде:

```
$fido = new Camel "Amelia";
```

то создавали объект `Camel`, представляемый скаляром `$fido`. Но внутреннее строение `Camel` является более сложным. Как воспитанные объектно-ориентированные программисты, мы не должны интересоваться содержимым

объектов Camel (если только перед нами не поставлена задача реализации методов класса Camel). Но обычно объект типа Camel состоит из хеша, содержащего атрибуты конкретного объекта Camel, такие как кличка (в данном случае "Amelia", а не "fido") и количество горбов (которое мы не задали, но по умолчанию, вероятно, установлен один: смотри обложку).

Простые вещи

Если у вас не закружилась слегка голова после чтения предыдущего параграфа, значит ваша голова необычная. Обычно люди не любят иметь дело со сложными структурами данных, будь они правительственными или генеалогическими. Поэтому в естественных языках есть много способов сделать вид, что никаких сложностей нет. Многие из них относятся к категории *задания темы* (*topicalization*), что служит причудливым лингвистическим термином, означающим соглашение с кем-то относительно того, о чем пойдет речь (или путем исключения, о чем речь, вероятно, не пойдет). В языке это происходит на нескольких уровнях. На верхнем уровне мы разделяемся на различные субкультуры, заинтересованные в различных подтемах и устанавливающие субъязыки, говорящие преимущественно об этих подтемах. Жаргон медицинской клиники («нерастворимый фактор асфиксии» – indissoluble asphyxiant) отличен от жаргона шоколадной фабрики («долгоиграющий гобстоппер» – everlasting gobstopper[1]). Большинство из нас автоматически переключает контекст при переходе с одного жаргона на другой.

На разговорном уровне переключение контекста должно производиться более явным образом, поэтому в нашем языке есть много способов сообщить, о чем мы собираемся говорить. Мы озаглавливаем книги и параграфы, из которых они состоят. В свои предложения мы вставляем витиеватые фразы типа «по поводу вашего последнего запроса» или «для каждого X». Обычно, однако, мы просто произносим нечто вроде: «Ну, знаешь, эта висюлька, которая болтается у тебя на шее?»

В Perl есть несколько возможностей задать тему. Одним из важных способов является объявление пакета (package declaration). Предположим, что мы хотим поговорить на Perl о верблюдах. Скорее всего, следует начать модуль Camel словами:

```
package Camel;
```

Это влечет несколько примечательных последствий. Одно из них состоит в том, что, начиная с этого места, Perl будет считать все неопределенные глаголы и существительные относящимися к Camel. Он делает это, автоматически снабжая префиксом с именем модуля Camel:: все глобальные имена. Поэтому если сказать:

```
package Camel;
$fido = &fetch();
```

[1] Большая многослойная конфета. – *Примеч. перев.*

то настоящим именем `$fido` будет `$Camel::fido` (а настоящим именем `&fetch` будет `&Camel::fetch`, но пока мы не будем говорить о глаголах). Это означает, что если в другом модуле сказано:

```
package Dog;
$fido = &fetch();
```

то Perl не напутает, поскольку настоящим именем этой `$fido` будет `$Dog::fido`, а не `$Camel::fido`. «По-научному» говорится, что пакет устанавливает *пространство имен* (*namespace*). Количество пространств имен не ограничено, но поскольку одновременно нельзя находиться сразу в нескольких, то можно притвориться, что остальных пространств имен не существует. Благодаря этому пространства имен упрощают жизнь. Упрощение основывается на притворстве. (Так же, конечно, как и чрезмерное упрощение, которым мы занимаемся в данной главе.)

Содержать в порядке существительные важно, но так же важно содержать в порядке глаголы. Прекрасно, что `&Camel::fetch` нельзя спутать с `&Dog::fetch` в пространствах имен `Camel` и `Dog`, но действительно замечательно в пакетах то, что они классифицируют глаголы так, что ими могут пользоваться *другие* пакеты. Когда мы говорим:

```
$fido = new Camel "Amelia";
```

то фактически вызываем глагол `&new` в пакете `Camel`, полным именем которого является `&Camel::new`. А когда мы говорим:

```
$fido->saddle();
```

то вызываем метод `&Camel::saddle`, поскольку переменная `$fido` помнит, что она указывает на `Camel`. Вот так действует объектно-ориентированное программирование.

Словами `package Camel` мы начинаем новый пакет. Но иногда требуется взять существительные и глаголы из уже существующего пакета. Perl позволяет сделать это посредством объявления `use` (использовать), которое обеспечивает не только заимствование глаголов из другого пакета, но и загрузку указанного модуля с диска. На деле, *необходимо* сказать что-то вроде:

```
use Camel;
```

перед тем, как сказать:

```
$fido = new Camel "Amelia";
```

потому что иначе Perl не будет знать, что такое `Camel`.

Интересно, что разработчику, в сущности, не требуется знать, что такое `Camel`, при условии, что кто-то другой может написать модуль `Camel` вместо него. Еще лучше, если кто-то *уже* написал модуль `Camel`. Можно утверждать, что мощь Perl заключается не в самом Perl, а в CPAN (Comprehensive Perl Archive Network) – архиве, содержащем мириады модулей, выполняющих

многочисленные задачи, которые вы сами не знаете, как решить. Вам нужно лишь загрузить любой понравившийся модуль и сказать:

```
use Some::Cool::Module;
```

После этого можно использовать глаголы из этого модуля так, как описывается в излагаемой здесь теме.

Подобно тому как это происходит в естественном языке, установка темы в Perl «искажает» язык, который будет использоваться с данного места и до конца программы. На практике некоторые встроенные модули вообще не вводят в действие новые глаголы, а только деформируют язык Perl разными полезными способами. Эти особые модули мы называем *прагмами* (*pragmas*). Например, часто применяется прагма `strict`:

```
use strict;
```

Действие модуля `strict` состоит в ужесточении некоторых правил, в результате чего чаще приходится указывать различные вещи, о которых в другом случае Perl догадывался бы сам, например, об областях видимости переменных. Явные указания выгодны при работе над большими проектами. По умолчанию Perl оптимизирован для маленьких проектов, но в случае применения прагмы `strict` хорошо использовать Perl и в больших проектах, требующих лучшего сопровождения. Поскольку прагму `strict` можно добавить в любой момент, Perl хорошо справляется с разрастанием маленьких проектов в большие, даже если поначалу это не предполагалось. Как обычно и бывает.

Глаголы

Как это свойственно и обычным «повелительным» языкам программирования, многие глаголы Perl являются командами: они приказывают интерпретатору Perl что-либо выполнить. С другой стороны, что характерно для естественных языков, значения глаголов Perl имеют тенденцию «разбегаться» в разных направлениях в зависимости от контекста. Высказывание, начинающееся с глагола, обычно является чисто повелительным и выполняется исключительно ради своих побочных эффектов. (Иногда мы называем такие глаголы *процедурами* (*procedures*), особенно если они определяются пользователем.) К часто встречающимся встроенным командам относится команда `print` (мы ее уже видели):

```
print "Женой Адама является $wife{'Адам'}.\n";
```

Ее побочный эффект выражается в появлении желаемого вывода:

```
Женой Адама является Ева.
```

Но кроме повелительного существуют и другие «наклонения». Некоторые глаголы служат для задания запросов и полезны в условных операторах, таких как оператор `if`. Другие глаголы транслируют свои входные параметры

в возвращаемые значения, подобно тому как рецепт описывает способ превращения исходных ингредиентов в нечто (желательно) съедобное. Мы склонны называть такие глаголы *функциями* (*functions*) из уважения к поколениям математиков, которым неизвестно значение слова «функциональный» в обычном языке.

Примером встроенной функции служит экспоненциальная функция:

```
$e = exp(1);   # 2.718281828459 или около того
```

Однако в Perl не проводится жесткого разделения между процедурами и функциями. Читатель обнаружит, что эти термины используются взаимозаменяемым образом. Иногда глаголы называются также операторами (если они встроенные) или подпрограммами (если они определены пользователем).[1] Но назовите их как хотите – все они возвращают значение, которое может быть осмысленным или нет и которое можно игнорировать или нет.

По мере нашего изложения вы увидите и другие примеры того, как Perl ведет себя аналогично естественным языкам. Но можно взглянуть на Perl и с других точек зрения. Мы уже «втихомолку» ввели некоторые обозначения из языка математики, такие как индексы, сложение, экспоненциальная функция. Но Perl является также языком управления, связующим (glue) языком, языком создания прототипов, языком обработки текста, обработки списков и объектно-ориентированным языком – помимо всего прочего.

Perl – это также старый добрый язык программирования (plain old computer language). С этой точки зрения мы и рассмотрим его теперь.

Стандартный пример

Предположим, что вы преподаете Perl группе студентов и хотите решить, как их оценивать. У вас есть набор результатов сдачи экзаменов каждым студентом, составленный в произвольном порядке. Требуется получить список, объединяющий оценки каждого студента и дополненный средним баллом. И у вас есть текстовый файл (оригинально названный *grades* – оценки), который выглядит так:

---

[1] Исторически Perl требовал указания символа «амперсанда» (&) при любых вызовах подпрограмм, определенных пользователем (смотри выше `$fido = &fetch();`). Но в Perl версии 5 задание амперсанда стало необязательным, поэтому при обращении к глаголам, определенным пользователем, теперь допускается такой же синтаксис, как и со встроенными глаголами (`$fido = fetch();`). Мы по-прежнему используем амперсанд, говоря об *имени* подпрограммы, например, при получении ссылки на нее (`$fetcher = \&fetch;`). С позиций лингвистики можно рассматривать формат `&fetch` с амперсандом как инфинитив, «to fetch», или аналогичную форму «do fetch». Но мы редко говорим «do fetch», если можно сказать просто «fetch». Это и есть действительная причина того, что мы отбросили обязательный амперсанд в Perl 5.

```
Nolл 25
Ben 76
Clementine 49
Norm 66
Chris 92
Doug 42
Carol 25
Ben 12
Clementine 0
Norm 66
...
```

С помощью следующего сценария можно собрать вместе все оценки, определить средний балл для каждого студента и вывести все это в алфавитном порядке. В программе наивно предполагается, что в группе нет двух студенток с именем Carol. А именно, если есть запись, относящаяся ко второй Carol, программа решит, что это еще одна оценка для первой Carol (не путать с первой Noel).

Кстати, номера строк не являются частью программы, несмотря на возможное сходство с BASIC в другом.

```
 1  #!/usr/bin/perl
 2
 3  open(GRADES, "grades") or die "Невозможно открыть grades: $!\n";
 4  while ($line = <GRADES>) {
 5      ($student, $grade) = split(" ", $line);
 6      $grades{$student} .= $grade . " ";
 7  }
 8
 9  foreach $student (sort keys %grades) {
10      $scores = 0;
11      $total = 0;
12      @grades = split(" ", $grades{$student});
13      foreach $grade (@grades) {
14          $total += $grade;
15          $scores++;
16      }
17      $average = $total / $scores;
18      print "$student: $grades{$student}\tСреднее: $average\n";
19  }
```

Пока у вас не разбежались глаза, мы лучше сразу отметим, что в этом примере демонстрируется многое из того, о чем мы уже рассказывали, а также многие вещи, которые мы объясним вскоре. Но если взглянуть на этот пример несколько издалека, то можно обнаружить некоторые интересные конструкции. Попробуйте сами догадаться о том, что здесь происходит, а позднее мы скажем, подтвердились ли эти догадки.

Мы бы посоветовали вам попробовать запустить этот пример на выполнение, но вы, вероятно, еще не знаете, как это делается.

Как это делается

Ну, сейчас вы, наверное, размышляете, как же запустить программу Perl. Короткий ответ состоит в том, что нужно подать ее на вход интерпретатора языка Perl, который, по счастливому совпадению, носит название *perl*. Длинный ответ начинается так: «There's More Than One Way To Do It» – есть несколько способов сделать это.[1]

Первый способ запустить *perl* (как правило, работающий в любой операционной системе) – это просто вызвать *perl* явным образом из командной строки.[2] Если вы выполняете что-то очень простое, можно использовать ключ `-e` (в следующем примере символ `%` представляет стандартное приглашение оболочки, поэтому его вводить не надо). В Unix можно ввести:

```
% perl -e 'print "Hello, world!\n";'
```

В других операционных системах могут понадобиться некоторые махинации с кавычками. Но базовый принцип остается одним и тем же: нужно попытаться уместить все, что требуется сообщить Perl, примерно в 80 колонок.[3]

Для создания более длинных сценариев можно с помощью своего любимого текстового редактора (или любого другого текстового редактора) поместить все команды в файл, а затем, предполагая, что вы назвали этот файл *gradation*, сказать:

```
% perl gradation
```

При этом по-прежнему интерпретатор Perl вызывается явным образом, но, по крайней мере, не приходится каждый раз вводить все в командной строке и не нужно возиться с кавычками, чтобы удовлетворить требованиям оболочки.

Удобнее всего вызвать сценарий, просто указав его имя (или щелкнув по нему) и предоставив операционной системе самой найти требуемый интерпретатор. В некоторых системах существуют способы связывания определенных расширений файлов или каталогов с определенными приложениями. В этих системах нужно произвести действия, необходимые для связывания

---

[1] Это девиз Perl, и вам неоднократно придется слышать его, если только вы не являетесь местным экспертом, и тогда вам неоднократно придется его произносить. Иногда его сокращают до TMTOWTDI, что произносится «Тим Тоуди». Но каждый волен произносить это так, как ему больше нравится. В конце концов, TMTOWTDI.

[2] В предположении, что ваша система имеет интерфейс командной строки. Тем, кто является обладателем достаточно старого Mac, возможно, потребуется обновить операционную систему до какой-либо версии BSD, такой как Mac OS X.

[3] Такого типа сценарии часто называют «однострочниками» («one-liners»). Попав в компанию других программистов Perl, вы обнаружите, что некоторые из них очень увлекаются созданием замысловатых однострочников. Из-за этих махинаций злые языки иногда называют Perl языком «только для записи».

сценария Perl с интерпретатором *perl*. В системах Unix, поддерживающих нотацию `#!` (нотация «shebang»), а таковыми в наше время является большинство систем Unix, можно с помощью специальной первой строки сценария указать операционной системе, какую программу она должна запустить. Введите в качестве первой строки нашего примера нечто аналогичное

```
#!/usr/bin/perl
```

(Если *perl* находится не в */usr/bin*, то нужно соответствующим образом изменить строку `#!`.) После этого вам требуется лишь сказать:

```
% gradation
```

Разумеется, это не сработало, поскольку вы забыли сделать сценарий исполняемым модулем (см. страницу руководства по *chmod*(1)) и поместить его в свой маршрут поиска PATH. Если его нет в переменной PATH, необходимо задать полное имя файла, чтобы операционная система знала, где найти ваш сценарий, например так:

```
% /home/sharon/bin/gradation
```

Наконец, если вам настолько не везет, что у вас стоит древняя система Unix, которая не поддерживает магическую строку `#!`, или путь к вашему интерпретатору длиннее 32 символов (предел, устанавливаемый на многих системах), вы можете обойти эти ограничения таким способом:

```
#!/bin/sh - - # perl, остановить цикл
eval 'exec /usr/bin/perl -S $0 ${1+"$@"}'
    if 0;
```

В некоторых операционных системах требуется какая-нибудь разновидность этого приема, чтобы справиться с */bin/csh*, *DCL*, *COMMAND.COM* или другим интерпретатором командной строки, устанавливаемым по умолчанию. Посоветуйтесь с Местным Экспертом.

На протяжении всей этой книги мы будем использовать просто `#!/usr/bin/perl` для представления всех этих обозначений, но теперь вы знаете, что под этим имеется в виду.

Случайный совет: когда будете писать контрольный сценарий, не называйте его *test*. В системах Unix имеется встроенная команда *test*, которая, скорее всего, и будет выполнена вместо вашего сценария. Попробуйте лучше назвать сценарий *try*.

А вот не столь случайный совет: изучая Perl, даже если вам кажутся понятными ваши действия, используйте ключ `-w`, особенно в процессе разработки. С помощью этой опции включается вывод всевозможных полезных и интересных предупредительных сообщений (не обязательно именно в такой очередности). Ключ `-w` можно указать и в первой строке:

```
#!/usr/bin/perl -w
```

Теперь, когда вы знаете, как запустить свою собственную программу Perl (не путать с программой *perl*), вернемся к нашему примеру.

Дескрипторы файлов

Если только вы не занимаетесь моделированием философа-солипсиста с помощью методов искусственного интеллекта, вашей программе требуются какие-то средства общения с внешним миром. В строках 3 и 4 нашего Стандартного примера есть слово `GRADES`, которое иллюстрирует еще один из типов данных Perl, а именно *дескриптор файла* (*filehandle*). Дескриптор файла – это просто имя, которое присваивается файлу, устройству, сокету или конвейеру и которое просто помогает вспомнить, о чем из перечисленного вы говорите, а также скрыть некоторые сложные вещи типа буферизации. (Во внутренней реализации дескрипторы файлов аналогичны потокам (streams) языков типа C++ или каналам ввода/вывода в BASIC.)

Дескрипторы файлов облегчают получение входных данных и отправку выходных в различные места. Хорошим связующим языком (glue language) делает Perl отчасти то, что он может одновременно общаться с многими файлами и процессами. Наличие удобных символических имен для различных внешних объектов является частью того, что составляет хороший связующий язык.[1]

Создание дескриптора файла и закрепление его за файлом выполняется с помощью функции `open`. Функция `open` принимает по меньшей мере два параметра: дескриптор файла и имя файла, с которым его нужно связать. Perl предоставляет также несколько предварительно определенных (и предварительно открытых) дескрипторов файлов. `STDIN` – это обычный канал ввода для вашей программы, а `STDOUT` – это для нее обычный канал вывода. `STDERR` представляет собой дополнительный канал вывода, позволяющий программе отпускать ехидные ремарки, пока она преобразует (или пытается преобразовать) ваши входные данные в выходные.[2]

---

[1] Вот и еще аспекты, благодаря которым Perl является хорошим связующим языком: он использует все 8 разрядов, его можно встраивать, а с помощью модулей расширения в него можно встраивать другие объекты. Он лаконичен, и с его помощью легко осуществлять сетевое взаимодействие. Он, можно сказать, чувствует окружающую среду. Он может вызываться различными способами (как было показано выше). Но главное, сам язык не настолько строго структурирован, чтобы нельзя было каким-то способом решить стоящую перед вами проблему. Здесь снова проявляется знакомый уже нам принцип TMTOWTDI.

[2] Обычно эти дескрипторы файлов привязаны к терминалу, чтобы можно было ввести данные с клавиатуры и посмотреть результат на экране, но можно связать их и с файлами (или чем-то еще). Perl обеспечивает эти предопределенные дескрипторы, поскольку они уже тем или иным способом обеспечены операционной системой. В Unix процессы наследуют стандартные устройства ввода, вывода и вывода ошибок от родительского процесса, которым обычно является оболочка. Одной из задач оболочки является настройка этих потоков ввода/вывода таким образом, чтобы дочернему процессу не пришлось о них беспокоиться.

Поскольку посредством функции open можно открывать дескрипторы файлов в различных целях (для ввода, вывода, конвейеризации), необходимо иметь возможность указать требуемый режим. Аналогично тому, как это делается в командной строке, нужно просто добавить символы к имени файла.

```
open(SESAME, "имя_файла")                # чтение из имеющегося файла
open(SESAME, "<имя_файла")               #   (то же, явным образом)
open(SESAME, ">имя_файла")               # создать файл и производить в него запись
open(SESAME, ">>имя_файла")              # дописывание к существующему файлу
open(SESAME, "| выходная_программа_конвейера")  # организовать выходной фильтр
open(SESAME, "входная_программа_конвейера |")   # организовать входной фильтр
```

Как можно видеть, имя для дескриптора файла выбирается произвольно. После открытия дескриптор файла SESAME можно использовать для доступа к файлу или конвейеру, пока он не будет явно закрыт (с помощью, как можно догадаться, close(SESAME)) либо пока дескриптор файла не будет связан с новым файлом посредством нового вызова open с тем же самым дескриптором.[1]

Открыв дескриптор файла для ввода, можно прочесть строку с помощью оператора чтения строки, <>. Его называют также оператором угловых скобок (angle operator). В оператор угловых скобок заключается имя дескриптора файла (<SESAME>), строки которого требуется читать. Пустой оператор угловых скобок <> читает строки из всех файлов, указанных в командной строке, или STDIN, если файлы не заданы. (Это стандартный режим для многих программ фильтрации.) Пример использования дескриптора файла STDIN для чтения данных, вводимых оператором, может выглядеть примерно так:

```
print STDOUT "Введите число: ";            # запрос числа
$number = <STDIN>;                         # ввод числа
print STDOUT "Вы ввели число $number.\n";  # вывод числа
```

Заметили, что мы только что вам подкинули? Что там делает STDOUT в этих командах print? Это всего лишь один из способов использования дескриптора файла выходных данных. Дескриптор файла можно передать в качестве первого параметра команды print, и если он указан, то определяет, куда должна быть направлена выдача. В данном случае указание дескриптора файла излишне, поскольку и без него выдача была бы направлена на STDOUT. Совсем как STDIN устанавливается по умолчанию для ввода, STDOUT устанавливается по умолчанию для вывода. (В строке 18 нашего Стандартного примера мы опустили его, чтобы не привести вас раньше времени в замешательство.)

---

[1] Открытие ранее открытого дескриптора файла приводит к неявному закрытию первоначального файла, который становится недоступным дескриптору, и открытию нового файла. Необходимо проявлять осторожность – действительно ли это то, что вы собираетесь делать. Иногда это происходит случайно, например, если сказать open($handle,$file), а $handle содержит ту же строку. Следите, чтобы $handle указывал на уникальную строку, иначе будет открыт новый файл с прежним дескриптором. Можно также оставить $handle неопределенным, а Perl сам даст ему значение.

Запустив предыдущий пример на исполнение, можно заметить, что выводится лишняя пустая строка. Это происходит потому, что операция чтения строки не удаляет автоматически символ перевода строки из строки ввода (пользователь мог ввести, например, "9\n"). Для тех случаев, когда нужно удалить символ перевода строки, в Perl есть функции chop и chomp. Функция chop удаляет без разбора (и возвращает) последний символ строки, в то время как chomp удаляет только маркер конца записи (обычно "\n") и возвращает число удаленных символов. Часто можно видеть такую идиому для ввода отдельной строки:

```
chop($number = <STDIN>);     # ввести число и удалить перевод строки
```

что равносильно следующему:

```
$number = <STDIN>;           # ввести число
chop($number);               # удалить перевод строки
```

Операторы

Как уже упоминалось выше, Perl является и математическим языком. Это справедливо для нескольких уровней: от нижнего уровня поразрядных логических операций на уровне разрядов, вверх до операций над числами и множествами и вплоть до больших предикатов и абстракций разного типа. А как нам известно из школьного курса математики, в ней любят употреблять необычные символы. Хуже, что специалисты по вычислительной технике придумали свои варианты этих странных символов. В Perl тоже есть несколько таких странных символов, но, по правде сказать, в большинстве своем они непосредственно заимствованы из C, FORTRAN, *sed*(1) или *awk*(1), поэтому пользователям этих языков они должны быть знакомы.

Остальные могут утешиться тем, что изучение этих странных символов в Perl может послужить началом изучения всех этих странных языков.

Встроенные операторы Perl могут быть классифицированы по числу операндов как унарные, бинарные и тернарные операторы. Их можно классифицировать по тому, являются ли они префиксными (предшествующими своим операндам) или инфиксными (указываемыми между своими операндами). Можно также классифицировать их по типу объектов, с которыми они работают, например, с числами, строками, файлами. Далее мы приведем таблицу со всеми операторами, но для начала приведем несколько полезных операторов.

Некоторые бинарные арифметические операторы

Арифметические операторы выполняют то, что и можно от них ожидать, если вы изучали их в школе. Они осуществляют некоторые математические функции над числами, например:

Пример	Название	Результат
`$a + $b`	Сложение	Сумма `$a` и `$b`
`$a * $b`	Умножение	Произведение `$a` и `$b`
`$a % $b`	Взятие по модулю	Остаток от деления `$a` на `$b`
`$a ** $b`	Возведение в степень	`$a` в степени `$b`

Да, мы опустили вычитание и деление – полагаем, что читатель сообразит, как они действуют. Поработайте с ними и проверьте, правы ли вы. (Или смошенничайте и посмотрите в главе 3 «Унарные и бинарные операторы».) Арифметические операторы выполняются в том порядке, которому вас учил школьный учитель математики (возведение в степень раньше умножения; умножение раньше сложения). Для изменения очередности всегда можно воспользоваться скобками.

Строковые операторы

Существует также оператор «сложения» для строк, осуществляющий конкатенацию (т. е. соединяющий концы строк). В отличие от языков, в которых эта операция выглядит так же, как сложение чисел, в Perl для конкатенации строк определен отдельный оператор (`.`):

```
$a = 123;
$b = 456;
print $a + $b;     # выводит 579
print $a . $b;     # выводит 123456
```

Есть также оператор «умножения» для строк, называемый оператором *повторения* (*repeat*). Опять же, это самостоятельный оператор (`x`), отличный от умножения чисел:

```
$a = 123;
$b = 3;
print $a * $b;     # выводит 369
print $a x $b;     # выводит 123123123
```

Эти строковые операторы присваивают значения столь же надежно, как соответствующие им арифметические операторы. Оператор повторения несколько необычен тем, что принимает строку в качестве левого аргумента и число в качестве правого. Обратите также внимание, что Perl автоматически преобразует числа в строки. Все приведенные выше числа-литералы можно было бы заключить в кавычки, и выдача от этого не изменилась бы. Внутренне, однако, преобразование производилось бы в обратном направлении, т. е. из строк в числа.

Следует обратить внимание еще на некоторые вещи. К конкатенации строк приводит также интерполяция, происходящая в строках, заключенных в двойные кавычки. И при выводе списка значений фактически производится

конкатенация строк. В результате три следующие команды дают одинаковый вывод:

```
print $a . ' равно ' . $b . ".\n";     # оператор "точка"
print $a, ' равно ', $b, ".\n";        # список
print "$a равно $b.\n";                # интерполяция
```

Которую из них использовать в конкретной ситуации, зависит целиком от вас. (Однако имейте в виду, что интерполяция часто читается легче всего.)

Оператор `x` на первый взгляд кажется относительно бесполезным, но иногда он очень удобен, особенно для таких вещей:

```
print "-" x $scrwid, "\n";
```

В результате поперек экрана вычерчивается линия, если только `$scrwid` содержит ширину экрана (*scr*een *wid*th), а не идентификатор вашего штопора (*screw id*entifier).

Операторы присваивания

Хотя этот оператор является не совсем математическим, мы уже широко пользовались простым оператором присваивания `=`. Постарайтесь запомнить, что `=` означает «устанавливается равным», а не «равно». (Существует и математический оператор равенства `==`, который означает «равно», и если вы с самого начала усвоите разницу между ними, то убережетесь от возникновения проблем в будущем. Оператор `==` действует как функция, возвращающая булево значение, тогда как `=` больше похож на процедуру, которая выполняется для получения побочного эффекта модификации переменной.)

Как и операторы, описанные выше, операторы присваивания представляют собой бинарные инфиксные операторы; это значит, что их операнды расположены по обеим сторонам оператора. Правый операнд может быть произвольным выражением, но левый должен быть допустимым *l-значением*[1] (*lvalue*), что в переводе на обычный язык означает допустимую область памяти типа переменной или элемента массива. Чаще всего используется простой оператор присваивания. Он вычисляет значение выражения, находящегося в правой части, а затем делает переменную в левой части равной этому значению:

```
$a = $b;
$a = $b + 5;
$a = $a * 3;
```

Заметьте, что последнее присваивание дважды ссылается на одну и ту же переменную: один раз при вычислении, и один раз при присваивании. В этом нет ошибки, но такая операция выполняется столь часто, что для нее существует сокращенное обозначение (заимствованное из C). Операция вида:

---

[1] Термин lvalue (от left value – левое значение) берет начало от оператора присваивания E1 = E2, где левый аргумент должен быть l-значением. – *Примеч. науч. ред.*

```
l-значение оператор= выражение
```

выполняется так, как если бы было записано:

```
l-значение = l-значение оператор выражение
```

за исключением того, что l-значение не вычисляется дважды. (Разница есть лишь тогда, когда вычисление l-значения имеет побочные эффекты. Но в тех случаях, когда разница *есть*, она обычно состоит именно в том, что вам требуется. Так что не стоит об этом волноваться.)

Так, предыдущий пример можно было бы записать в следующем виде:

```
$a *= 3;
```

что следует читать как «умножить $a на 3». Такое сокращение допустимо в Perl почти для любого бинарного оператора, даже для некоторых операторов, с которыми в C это нельзя делать:

```
$line .= "\n";  # Дописать символ перевода строки к $line.
$fill x= 80;    # Сделать строку $fill 80-кратным повторением самой себя.
$val ||= "2";   # Установить значение $val равным 2, если оно не "true".
```

В строке 6 нашего Стандартного примера[1] есть две конкатенации строк, одна из которых представляет собой оператор присваивания. А строка 14 содержит сокращенный оператор +=.

Независимо от того, какого типа оператор присваивания используется, в качестве значения присваивания в целом возвращается конечное значение переменной в левой части.[2] Это не удивит программистов на C, которым известно, как использовать эту идиому для обнуления переменных:

```
$a = $b = $c = 0;
```

Также часто можно видеть, что присваивание используется в условии цикла while, как в строке 4 нашего Стандартного примера.

Что *должно* удивить C-программистов, так это то, что присваивание в Perl возвращает фактическую переменную, выступающую как l-значение, поэтому можно модифицировать одну и ту же переменную несколько раз в одной команде. Например, можно написать:

```
($temp -= 32) *= 5/9;
```

для преобразования по месту из шкалы Фаренгейта в шкалу Цельсия. Это еще одна причина, по которой ранее в этой главе мы могли сказать:

```
chop($number = <STDIN>);
```

---

[1] А вы думали, что мы про него забыли, не так ли?

[2] Это не так, скажем, в Pascal, в котором присваивание является командой и не возвращает никакого значения. Мы говорили выше, что присваивание аналогично процедуре, но запомните, что в Perl даже процедуры возвращают значения.

чтобы «обрубить» конечное значение `$number`. Вообще говоря, эту особенность можно использовать всякий раз, когда требуется что-то скопировать и одновременно произвести над ним какие-то действия.

Унарные арифметические операторы

Как если бы запись `$variable += 1` была недостаточно короткой, Perl заимствует из C еще более краткий способ приращения переменной. Операторы инкремента (и декремента) просто прибавляют (или вычитают) единицу к значению переменной. Они могут помещаться с любой стороны переменной, в зависимости от того, в какой момент требуется их выполнить:

Пример	Название	Результат
`++$a, $a++`	Инкремент	Прибавить 1 к `$a`
`- -$a, $a- -`	Декремент	Вычесть 1 из `$a`

Когда один из этих «автоматических» операторов предшествует переменной, то это называется префиксным инкрементированием (декрементированием) переменной, ее значение будет изменено перед тем, как произойдет обращение к ней. Когда оператор помещается после переменной, то это называется постфиксным инкрементированием (декрементированием) переменной, и ее значение изменяется после использования. Например:

```
$a = 5;        # $a присваивается значение 5
$b = ++$a;     # $b присваивается инкрементированное значение $a, 6
$c = $a--;     # $c присваивается 6, затем $a декрементируется и становится 5
```

В строке 15 нашего Стандартного примера число результатов увеличивается на единицу, и мы получаем количество результатов, которые нужно усреднить. При этом используется оператор постфиксного инкрементирования (`$scores++`), но в данном случае это несущественно, поскольку выражение применяется в пустом контексте, что представляет собой лишь причудливый способ сказать, что выражение вычисляется только ради побочного эффекта инкрементирования переменной. Возвращаемое значение при этом отбрасывается.[1]

Логические операторы

Логические операторы, которые называют также операторами «короткого действия» («short-circuit»), позволяют программе принимать решения, исходя из нескольких критериев, без применения вложенных операторов `if`. Их называют операторами короткого действия, поскольку они пропускают

[1] Оптимизатор обратит на это внимание и преобразует постфиксное инкрементирование в префиксное, поскольку последнее выполняется немного быстрее. (Вам необязательно знать об этом, но мы надеялись, что это вас ободрит.)

(закорачивают) вычисление правого аргумента, если считают, что в левом аргументе содержится достаточно информации для получения итогового результата. Это делается не только для повышения производительности. Программист может явным образом использовать этот режим закорачивания, чтобы избежать вычислений в правом аргументе, о которых ему известно, что они сорвались бы, если бы их не «охранял» левый аргумент. В Perl можно сказать: «California or bust!» (в Калифорнию или провалиться), и при этом не провалиться (предполагая, что вы действительно доберетесь до Калифорнии).

На самом деле в Perl есть два набора логических операторов: традиционный набор, заимствованный из C, и более новый (но еще более традиционный) набор операторов со сверхнизким приоритетом, заимствованный из BASIC. Оба набора при надлежащем применении повышают удобочитаемость программ. Операторы C, использующие знаки пунктуации, уместны, когда желательно, чтобы логические операторы имели больший приоритет, чем запятые, тогда как записываемые словами операторы BASIC хорошо использовать, когда желательно, чтобы запятые имели больший приоритет, чем логические операторы. Часто они действуют одинаково, а использование того или иного набора является делом вкуса. (Примеры, выявляющие контраст, можно найти в разделе «Логические and, or, not и xor» главы 3 «Унарные и бинарные операторы».) Хотя из-за различных приоритетов эти два набора операторов не являются взаимозаменяемыми, после синтаксического анализа сами операторы ведут себя одинаково; приоритетность просто определяет область расположения их аргументов. В табл. 1.1 перечислены логические операторы.

Таблица 1.1. Логические операторы

Пример	Название	Результат
`$a && $b`	И	`$a`, если `$a` ложно, в противном случае `$b`
`$a || $b`	Или	`$a`, если `$a` истинно, в противном случае `$b`
`! $a`	Не	Истинно, если `$a` не истинно
`$a and $b`	И	`$a`, если `$a` ложно, `$b` в противном случае
`$a or $b`	Или	`$a`, если `$a` истинно, `$b` в противном случае
`not $a`	Не	Истинно, если `$a` не истинно
`$a xor $b`	Исключающее или	Истинно, если `$a` или `$b` истинны, но не оба сразу

Благодаря тому что логические операторы «закорачивают» вычисление, их часто используют в Perl для условного выполнения кода. В следующей строке (строка 3 нашего Стандартного примера) делается попытка открыть файл *grades*:

```
open(GRADES, "grades") or die "Невозможно открыть grades: $!\n";
```

Если файл удается открыть, происходит переход к следующей строке программы. Если файл открыть не удается, выводится сообщение об ошибке, и выполнение прекращается.

Буквально эта строка означает: «Открыть *grades* или умереть!» Операторы короткого действия служат еще одним примером естественности языка, но, кроме того, оберегают зрительное представление. Важные действия перечисляются в левой части экрана, а второстепенные скрыты справа. (Переменная `$!` содержит сообщение об ошибке, возвращенное операционной системой – см. главу 28 «Специальные идентификаторы».) Конечно, эти логические операторы можно использовать и в более традиционных условных выражениях, таких как операторы `if` и `while`.

Некоторые операторы сравнения чисел и строк

Операторы сравнения, или отношения, сообщают нам о том, как две скалярные величины (числа или строки) относятся одна к другой. Есть два набора операторов: один сравнивает числа, а другой – строки. (В любом случае аргументам будет сначала «принудительно» навязан соответствующий тип.) Предполагая, что левый и правый аргументы суть `$a` и `$b`, получаем:

Сравнение	Числа	Строки	Возвращаемое значение
Равно	`==`	`eq`	Истина, если `$a` равно `$b`
Не равно	`!=`	`ne`	Истина, если `$a` не равно `$b`
Меньше	`<`	`lt`	Истина, если `$a` меньше `$b`
Больше	`>`	`gt`	Истина, если `$a` больше `$b`
Больше или равно	`<=`	`le`	Истина, если `$a` не больше `$b`
Меньше или равно	`>=`	`ge`	Истина, если `$a` не меньше `$b`
Сравнение	`<=>`	`cmp`	0, если равно, 1, если `$a` больше, -1, если `$b` больше

Последние два оператора (`<=>` и `cmp`) совершенно избыточны. Однако они крайне полезны в подпрограммах сортировки `sort` (см. главу 29 «Функции»).[1]

Некоторые операторы проверки файлов

Операторы этой группы позволяют проверить атрибуты файлов, чтобы работать с этими файлами не вслепую. Основным атрибутом файла является, конечно, его существование. Например, весьма уместно поинтересоваться, не существует ли уже у вас файл с почтовыми псевдонимами, прежде чем от-

[1] Кое-кто воспринимает такую избыточность как порочную, поскольку из-за нее язык перестает быть «минималистским», или ортогональным. Но Perl и не является ортогональным языком, это диагональный язык. Мы подразумеваем под этим, что Perl не вынуждает программиста всегда перемещаться под прямым углом. Иногда хочется попасть в нужное место по гипотенузе треугольника. TMTOWTDI предполагает наличие сокращений. Сокращения предполагают эффективность.

крывать его как новый файл, удаляя при этом все имеющиеся в нем данные. Вот несколько операторов проверки файлов:

Пример	Название	Результат
`-e $a`	Существует (exists)	Истина, если файл с именем `$a` существует
`-r $a`	Доступен по чтению (readable)	Истина, если файл с именем `$a` доступен по чтению
`-w $a`	Доступен по записи (writable)	Истина, если файл с именем `$a` доступен по записи
`-d $a`	Каталог (directory)	Истина, если файл с именем `$a` является каталогом
`-f $a`	Файл (file)	Истина, если файл с именем `$a` является обычным файлом
`-T $a`	Текстовый файл (text file)	Истина, если файл с именем `$a` является текстовым файлом

Можно так использовать эти операторы:

```
-e "/usr/bin/perl" or warn "Perl неправильно установлен\n";
-f "/vmlinuz" and print "Вижу, что вы друг Лайнуса\n";
```

Обратите внимание, что обычный (regular) файл – не то же самое, что текстовый файл (text file). Двоичные файлы типа */vmlinuz* являются обычными, но не текстовыми. Текстовые файлы представляют собой противоположность двоичным файлам, тогда как обычные файлы – противоположность «иррегулярным» файлам типа каталогов или устройств.

Существует много операторов проверки файлов сверх перечисленных нами. Большинство таких операторов представляют собой унарные булевы операторы, т. е. принимают только один операнд (скаляр, значением которого является имя файла или его дескриптор) и возвращают значение true или false. Некоторые из них возвращают более замысловатые значения, например, размер или время, относящееся к файлу; их, когда они вам понадобятся, вы сможете найти в разделе «Именованные унарные операторы и операторы проверки файлов» в главе 3 «Унарные и бинарные операторы».

Управляющие структуры

До настоящего момента все рассмотренные нами примеры, за исключением одного большого примера, были совершенно прямолинейными: мы выполняли все команды по порядку. Было приведено несколько примеров использования операторов короткого действия для выполнения (или невыполнения) отдельной команды. Хотя можно писать очень полезные линейные программы (к которым относятся многие сценарии CGI), но при наличии условных выражений и циклов можно создавать значительно более мощные приложения. В совокупности эти механизмы называются управляющими

структурами (control structures). Поэтому можно рассматривать Perl и как управляющий язык.

Но для того чтобы управлять, необходимо иметь возможность принимать решения, а чтобы принимать решения, нужно знать, в чем разница между истиной и ложью.

Что есть истина?

Мы обсуждали термин «истинность» (truth)[1] и упоминали, что некоторые операторы возвращают значения true или false. Прежде чем двигаться дальше, необходимо точно договориться о том, что под ними подразумевается. В Perl значение истинности рассматривается несколько иначе, чем в других языках программирования, но, поработав с ним некоторое время, читатель найдет это очень разумным. (На самом деле, мы надеемся, что разумным это покажется вам после чтения нижеизложенного.)

В сущности, Perl считает, что истина самоочевидна. Это поверхностный способ сказать, что можно оценить истинность почти чего угодно. В Perl используется практическое определение истинности в зависимости от типа того, что подвергается оценке. Оказывается при этом, что типов истинности существует значительно больше, чем неистинности.

Истинность в Perl всегда вычисляется в скалярном контексте. Какого-либо другого приведения типов не производится. Вот правила для различных типов значений, которые может содержать скаляр:

1. Любая строка, кроме `""` и `"0"`, является истиной
2. Любое число, кроме 0, является истиной.
3. Любая ссылка является истиной.
4. Любая неопределенная величина является ложью.

Фактически последние два правила следуют из двух первых. Любая ссылка (правило 3) указывает на нечто, имеющее адрес, и имеет значением число или строку, содержащую этот адрес, которые не могут быть равны 0, поскольку всегда определены. А всякая неопределенная величина (правило 4) всегда имеет значением 0 или нулевую строку.

А в некотором смысле и правило 2 можно вывести из правила 1, если представить, что все является строками. Опять же, для вычисления истинности никакого приведения строк не производится, но если бы приведение строк *производилось*, то число 0 превратилось бы в строку `"0"` и оказалось бы ложью. Любое другое число превратилось бы в строку, отличную от `"0"`, а потому оказалось бы истиной. Рассмотрим несколько примеров, которые позволят лучше это понять:

```
0          # становится строкой "0", поэтому false.
1          # становится строкой "1", поэтому true.
```

[1] Строго говоря, это не истина.

```
10 - 10    # 10 минус 10 равно 0, преобразуется в строку "0", поэтому false.
0.00       # равно 0, преобразуется в строку  "0", поэтому false.
"0"        # является строкой "0", поэтому false.
""         # является нулевой строкой, поэтому false.
"0.00"     # является строкой "0.00", ни "", ни "0", поэтому true!
"0.00" + 0 # становится числом 0 (приведение из-за +), поэтому false.
\$a        # является ссылкой на $a, поэтому true, даже если $a есть false.
undef()    # функция, возвращающая неопределенное значение, поэтому false.
```

Поскольку ранее было что-то невнятно сказано о вычислении истинности в скалярном контексте, у читателя может возникнуть вопрос о том, как оценивается истинность списка. Дело в том, что ни одна операция Perl не возвращает список в скалярном контексте. Все они обнаруживают, что находятся в скалярном контексте, и потому возвращают скалярное значение, к которому и применяются правила истинности. Поэтому здесь не возникает проблемы, т. к. можно определить значение, возвращаемое любым данным оператором в скалярном контексте. Оказывается как массивы, так и хеши возвращают скалярные значения, являющиеся истиной, если в массиве или хеше есть хоть один элемент. Подробнее об этом далее.

Операторы if и unless

Мы видели выше, что логический оператор может действовать в качестве условного. Несколько более сложной формой логического оператора является оператор `if`. Оператор `if` вычисляет истинность условия (т. е. булево выражение) и выполняет блок кода, если условие истинно:

```
if ($debug_level > 0) {
    # Что-то не так. Сообщить пользователю.
    print "Отладка: Тревога, Уилл Робинсон, тревога!\n";
    print "Отладка: Ответ был '54', а ожидалось '42'.\n";
}
```

Блоком называется одна или несколько команд, заключенных в пару фигурных скобок. Поскольку оператор `if` выполняет блок, фигурные скобки требуются в нем по определению. Те, кто знаком с языком типа C, заметят различие. Скобки необязательно использовать в C, если команда всего одна, но в Perl скобки не являются факультативными.

Иногда недостаточно выполнить блок команд при удовлетворении условию. Может потребоваться выполнить другой блок, если условие *не* выполняется. Конечно, можно использовать два оператора `if` с условием одного, являющимся отрицанием условия другого, но Perl предоставляет более элегантное решение. После блока `if` может принять необязательное второе условие, называемое `else`, которое выполняется, только если условие ложно. (Ветеранов программирования это не должно удивить.)

Иногда возможных вариантов выбора оказывается больше двух. В таких случаях требуется добавить условие истинности `elsif` для каждого возможного варианта. (Ветеранов программирования может весьма удивить написание `elsif`, за что никто не собирается просить прощения. Извините.)

```
if ($city eq "New York") {
    print "Нью-Йорк находится на северо-востоке от Вашингтона, округ Колумбия.\n";
}
elsif ($city eq "Chicago") {
    print "Чикаго находится на северо-западе от Вашингтона, округ Колумбия.\n";
}
elsif ($city eq "Miami") {
    print "Майами находится на юге от Вашингтона. И там гораздо теплее!\n";
}
else {
    print "Извините, но я не знаю, где находится $city.\n";
}
```

Предложения `if` и `elsif` вычисляются по очереди, пока одно из них не окажется истинным или не будет достигнуто условие `else`. Когда одно из условий оказывается истинным, выполняется его блок, а все остальные ветви пропускаются. Иногда требуется, чтобы никакие действия не выполнялись, если условие истинно – только если оно ложно. Применение пустого `if` с `else` выглядит неряшливо, а `if` с отрицанием может быть трудным для чтения; в обычном языке фраза «если не это является истиной, сделать то-то и то-то» звучит несколько странно. В таких случаях следует предпочесть оператор `unless`:

```
unless ($destination eq $home) {
    print "Я еду не домой.\n";
}
```

Оператор `elsunless` отсутствует. Вообще говоря, его можно охарактеризовать как излишество (feature).

Итеративные (циклические) конструкции

В Perl есть четыре основных типа итеративных операторов: `while`, `until`, `for` и `foreach`. Эти операторы позволяют программе Perl повторять выполнение одного и того же кода.

Операторы while и until

Операторы `while` и `until` ведут себя аналогично операторам `if` и `unless`, но позволяют выполнять блок многократно. Иными словами, они выполняют цикл. Сначала проверяется значение части оператора, являющейся условием. Если условие соблюдено (т. е. оно принимает значение true для `while` или false для `until`), выполняется блок команд.

```
while ($tickets_sold < 10000) {
    $available = 10000 - $tickets_sold;
    print "$available билетов в наличии.  Сколько нужно вам: ";
    $purchase = <STDIN>;
    chomp($purchase);
    $tickets_sold += $purchase;
}
```

Обратите внимание, что если исходное условие не выполнено, вход в цикл вообще не производится. Например, если уже продано 10 000 билетов, можно потребовать, чтобы следующая строка программы сообщала что-то вроде:

```
print "Билеты на это представление распроданы, пожалуйста,
зайдите в следующий раз.\n";
```

В строке 4 нашего Стандартного примера осуществляется чтение:

```
while ($line = <GRADES>) {
```

В результате очередная строка присваивается переменной `$line` и, как объяснялось выше, возвращается значение `$line`, поэтому в условии оператора `while` может быть определена истинность `$line`. Читателю может показаться, что Perl получит ошибочное ложное значение при вводе пустой строки и досрочно выйдет из цикла. На самом деле этого не произойдет. Причина будет понятна, если поразмыслить обо всем сказанном выше. Оператор ввода строки оставляет в конце строки символ перевода строки, поэтому пустая строка имеет значение `"\n"`. А как известно, `"\n"` не является каким-либо из канонических ложных значений. Поэтому условие истинно, и цикл продолжается даже при вводе пустых строк.

С другой стороны, когда в конце концов будет достигнут конец файла, оператор ввода строки возвратит неопределенное значение, которое всегда вычисляется как «ложь». В результате выполнение цикла прекращается там, где это нам и нужно. В Perl не требуется явно проверять значение функции `eof`, поскольку операторы ввода разработаны так, чтобы гладко работать в контексте условия.

На практике почти все спроектировано так, чтобы гладко работать в условном (булевом) контексте. Если обратиться к массиву в скалярном контексте, то возвращается размер массива. Поэтому можно часто видеть такую обработку аргументов командной строки:

```
while (@ARGV) {
    process(shift @ARGV);
}
```

Оператор `shift` при каждой итерации цикла удаляет один элемент из списка аргументов (и возвращает его). Цикл автоматически завершается, когда исчерпывается массив `@ARGV`, т. е. его длина становится равной 0. А 0 в Perl является ложью. В некотором смысле сам массив становится ложью.[1]

---

[1] Это стиль мышления программистов на Perl. Поэтому не нужно сравнивать 0 и 0, чтобы проверить, не является ли значением этого выражения «ложь». Несмотря на то что в других языках это требуется, не вступайте на путь задания явных сравнений типа `while (@ARGV != 0)`. Это неэффективно как для вас, так и для компьютера. Да и для каждого, кому придется сопровождать ваш код.

Оператор for

Другой итеративный оператор – цикл `for`, выполняемый точно так же, как цикл `while`, но выглядящий совсем иначе. (Однако C-программистам он покажется очень знакомым.)

```
for ($sold = 0; $sold < 10000; $sold += $purchase) {
    $available = 10000 - $sold;
    print "$available билетов в наличии. Сколько нужно вам: ";
    $purchase = <STDIN>;
    chomp($purchase);
}
```

Цикл `for` принимает три выражения, заключенные в круглые скобки: выражение для начальной установки переменной цикла, условие для проверки переменной цикла и выражение для изменения состояния переменной цикла. В начале цикла устанавливается исходное значение и проверяется истинность условия. Если условие истинно, выполняется блок. Когда блок кончается, выполняется модифицирующее выражение, снова проверяется истинность условия, и, если условие выполнено, блок выполняется снова с новым значением переменной. Пока условие сохраняет истинность, будут выполняться блок и модифицирующее выражение. (Обратите внимание на то, что только среднее выражение вычисляется с целью получения его значения. Первое и третье выражения вычисляются только ради получения побочного эффекта, а их значения отбрасываются!)

Оператор foreach

Последним из итеративных операторов Perl является оператор `foreach`, применяемый для выполнения одного и того же кода для каждого скаляра из некоторого набора, которым может, например, быть массив:

```
foreach $user (@users) {
    if (-f "$home{$user}/.nexrc") {
        print "$user крут... он использует vi, поддерживающий perl!\n";
    }
}
```

В отличие от операторов `if` и `while`, которые обеспечивают скалярный контекст в условном выражении, оператор `foreach` обеспечивает списочный контекст для выражения в круглых скобках. Поэтому в результате вычисления выражения получается список (не скаляр, даже если в списке лишь один скаляр). После этого переменная цикла поочередно получает имена всех элементов списка, и блок кода выполняется по одному разу для каждого элемента списка. Обратите внимание на то, что переменная цикла ссылается на сам элемент, а не на его копию. В результате модификация переменной цикла приводит к модификации исходного массива.

В типичной программе Perl можно обнаружить значительно больше циклов `foreach`, чем циклов `for`, поскольку в Perl очень легко создавать такие спис-

ки, которые требуются foreach для итерации. Одной из идиом, которые встречаются часто, является цикл для перебора отсортированных ключей хеша:

```
foreach $key (sort keys %hash) {
```

На практике именно это делает строка 9 нашего Стандартного примера.

Выход из цикла: next и last

Операторы next и last позволяют изменять порядок выполнения команд в цикле. Не так уж редки особые ситуации, которые нужно пропускать или при возникновении которых нужно завершить цикл. Например, при работе с учетными записями в Unix может потребоваться пропустить такие учетные записи системы, как *root* или *lp*. Оператор next позволяет переместиться в конец текущей итерации цикла и начать новую итерацию. Оператор last позволяет перескочить конечную границу блока, как если бы условие выполнения цикла оказалось ложным. Это может быть полезным, если, к примеру, надо отыскать определенную учетную запись и, найдя ее, выйти из цикла.

```
foreach $user (@users) {
    if ($user eq "root" or $user eq "lp") {
        next;
    }
    if ($user eq "special") {
        print "Нашел учетную запись special.\n";
        # какая-то обработка
        last;
    }
}
```

Существует возможность прерывать выполнение вложенных циклов с помощью задания меток циклов и указания того цикла, из которого надо выйти. При использовании модификаторов операторов (другого вида условных операторов, о котором речь пойдет ниже) это позволяет создавать очень хорошо читаемые выходы из циклов (если вы случайно полагаете, что английский язык является хорошо читаемым):

```
LINE: while ($line = <ARTICLE>) {
    last LINE if $line eq "\n"; # остановиться на первой пустой строке
    next LINE if $line =~ /^#/; # пропустить строки комментариев
    # your ad here
}
```

Читатель может сказать: «Погодите, что это еще за нелепые значки ^# между наклонными зубочистками? Это не очень-то похоже на английский язык.» И будет прав. Это шаблон для поиска, содержащий регулярное выражение (хотя и довольно простое). О регулярных выражениях будет рассказано в следующем разделе. Perl является лучшим в мире языком обработки текста, а регулярные выражения образуют сердцевину обработки текста в Perl.

Регулярные выражения

Регулярные выражения (*regular expressions*) (известные также как regexes, regexps или RE) используются многими программами поиска, такими как *grep* и *findstr*, изменяющими текст («text-munging») программами, такими как *sed* и *awk*, а также редакторами, например *vi* и *emacs*. Регулярное выражение – это способ описать множество строк, не перечисляя все строки, входящие в это множество.[1]

Регулярные выражения входят во многие другие языки программирования (некоторые из них даже рекламируют «регулярные выражения Perl5»!), но ни в один из этих языков регулярные выражения не интегрированы так, как в Perl. Регулярные выражения используются в Perl несколькими способами. Во-первых, и это главное, они применяются в условных операторах, чтобы определить, соответствует ли строка некоторому шаблону, поскольку в булевом контексте они возвращают «истину» или «ложь». Поэтому, увидев в условном операторе нечто вроде `/foo/`, знайте, что перед вами обычный оператор *поиска по шаблону* (*pattern-matching*):

```
if (/Windows 95/) { print " Пора делать апгрейд?\n" }
```

Во-вторых, найдя шаблон в строке, можно заменить его чем-либо другим. Поэтому, увидев нечто типа `s/foo/bar/`, знайте, что это просьба Perl заменить, при возможности, «bar» на «foo». Мы называем это оператором *подстановки* (*substitution*). Он также возвращает true или false в зависимости от успешности операции, но обычно выполняется ради своего побочного результата:

```
s/Windows/Linux/;
```

Наконец, с помощью шаблонов можно задавать не только то, что где-то существует, но и то, чего *нет*. Поэтому оператор `split` использует регулярное выражение для задания того, где данных нет. Это значит, что регулярное выражение задает *разделители* (*separators*), разграничивающие поля данных. В нашем Стандартном примере есть пара простейших образцов. В строках 5 и 12 производится расщепление строк по пробелам и возвращается список слов. Но расщепление возможно и по любому другому разделителю, задаваемому регулярным выражением:

```
($good, $bad, $ugly) = split(/,/, "vi,emacs,teco");
```

(Есть разные модификаторы, которые можно применять в каждом из этих случаев для осуществления экзотических действий вроде игнорирования верхнего и нижнего регистра при поиске символов алфавита, но эти «кровавые подробности» мы будем освещать позже, когда доберемся до кровавых подробностей.)

---

[1] Хорошим источником информации по концепциям регулярных выражений служит книга Джеффри Фридла (Jeffrey Friedl) «Mastering Regular Expressions» (O'Reilly & Associates).

В простейшем случае регулярные выражения используются для поиска литерального выражения. В приведенном выше примере расщепления осуществлялся поиск одиночных запятых. Но если надо отыскать несколько символов подряд, то все они должны быть найдены один за другим. То есть шаблон, как и ожидалось, ищет подстроку. Допустим, что требуется показать все строки файла HTML, содержащие ссылки HTTP (но не ссылки FTP). Представим себе, что мы впервые работаем с файлом HTML и слегка наивны. Мы знаем, что в этих ссылках всегда где-то присутствует `http:`. Можно организовать такой цикл по нашему файлу:

```
while ($line = <FILE>) {
    if ($line =~ /http:/) {
        print $line;
    }
}
```

Здесь `=~` (оператор привязки шаблона) указывает Perl на необходимость поиска регулярного выражения `http:` в переменной `$line`. Если выражение будет найдено, то оператор возвратит значение «истина», и будет выполнен блок (оператор `print`).[1]

Между прочим, если не использовать оператор привязки `=~`, то Perl будет осуществлять поиск в строке, установленной по умолчанию, а не в `$line`. Это все равно, что сказать: «Ой! Помогите найти мою контактную линзу!» И люди станут автоматически искать ее около вас, хотя вы не говорили им, где нужно искать. Аналогично, Perl знает, что есть место, по умолчанию установленное для поиска, если не указано, где надо искать. Эта строка по умолчанию является на самом деле особой скалярной переменной с необычным именем `$_`. На практике это место по умолчанию не только для поиска по шаблону; многие операторы Perl используют по умолчанию переменную `$_`, поэтому программист Perl со стажем, вероятно, написал бы последний пример так:

```
while (<FILE>) {
    print if /http:/;
}
```

(Хм-м, похоже, что сюда проник еще один из этих модификаторов команд. Хитрые маленькие бестии.)

Все это достаточно хорошо, но что если нужно найти все типы ссылок, а не только ссылки HTTP? Можно дать список ссылок, например `http:`, `ftp:`, `mailto:` и т. д. Но этот список может оказаться длинным, и как быть, если появится новый тип ссылки?

---

[1] Это очень похоже на то, что делает команда Unix `grep 'http:' file`. В MS-DOS можно применять команду *find*, но она не умеет обрабатывать более сложные регулярные выражения. (Однако неудачно названная команда *findstr* в Windows NT умеет обрабатывать регулярные выражения).

```
while (<FILE>) {
    print if /http:/;
    print if /ftp:/;
    print if /mailto:/;
    # Что еще?
}
```

Поскольку регулярные выражения описывают множество строк, можно просто описать то, что мы ищем: несколько символов алфавита, за которыми следует двоеточие. На языке регулярных выражений это будет выглядеть как `/[a-zA-Z]+:/`, где скобки определяют *класс символов* (*character class*). Группы `a-z` и `A-Z` представляют все символы английского алфавита (черточка обозначает диапазон всех символов с начального по конечный включительно). А знак `+` является специальным символом, означающим «не менее одного из того, что мне предшествовало». Это то, что мы называем *квантификатором* (*quantifier*), представляющим собой штуковину, позволяющую сообщить, сколько раз нечто может повториться. (Символы косой черты, в сущности, являются не частью регулярного выражения, а частью оператора поиска по шаблону. Они действуют подобно кавычкам, в которые заключается регулярное выражение.)

Поскольку некоторые классы, например символы алфавита, используются очень часто, Perl определяет для них сокращения:

Название	Определение в ASCII	Код
Пробельный символ	`[ \t\n\r\f]`	`\s`
Символ слова	`[a-zA-Z_0-9]`	`\w`
Цифра	`[0-9]`	`\d`

Заметьте, что эти коды соответствуют *одиночным* символам. Метасимвол `\w` ищет соответствие любому одиночному символу слова, а не целому слову. (Квантификатор `+` еще не забыт? Можно задать `\w+` для поиска слова.) В Perl можно задать отрицание этих классов, используя символы на верхнем регистре, например, `\D` для символа, не являющегося цифрой.

Следует заметить, что `\w` не всегда эквивалентно `[a-zA-Z_0-9]` (а `\d` не всегда эквивалентно `[0-9]`). В некоторых наборах параметров локализации (locales) определяются дополнительные символы, не входящие в ASCII, которые соблюдаются метасимволом `\w`. Новым версиям Perl «знакомы» также свойства букв и цифр Unicode, и символы Unicode с этими свойствами обрабатываются надлежащим образом. (Perl и идеограммы рассматривает как символы `\w`.)

Есть еще один совершенно особенный класс символов, записываемый как точка (`.`), которому соответствует вообще любой символ.[1] Например, `/a./` соответствует любой строке, содержащей `a`, если только это не последний

[1] За исключением того, что обычно символ перевода строки ему не соответствует. Если вспомнить, то (`.`) обычно не соответствует переводу строки и в *grep*(1).

символ в строке. Таким образом, это выражение соответствует at или am, или даже a!, но не a, поскольку после a нет ничего, чему могла бы соответствовать точка. Поскольку поиск шаблона выполняется в любом месте строки, ему соответствуют также oasis и camel, но не sheba. Он соответствует caravan по первой a. Он мог бы соответствовать и второй a, но поиск останавливается после нахождения первого соответствия, слева направо.

Квантификаторы

Символы и классы символов, о которых мы говорили, ищут соответствие одиночным символам. Мы уже говорили, что можно искать соответствие нескольким символам «слова» посредством \w+. Одним из квантификаторов является +, но есть и другие. Все они помещаются после того элемента, количество вхождений которого определяют.

В самом общем виде квантификатор задает минимальное и максимальное допустимое число соответствий элемента. Эти два числа заключаются в скобки и разделяются запятой. Например, если пытаться найти номера телефонов в Северной Америке, то последовательность \d{7,11} соответствует не менее чем семи цифрам, но не более чем одиннадцати. Если в скобки помещено одно число, то оно задает одновременно минимум и максимум, т. е. точное число соответствий элемента. (Все элементы без указания количества имеют неявный квантификатор {1}.)

Если указать минимум и запятую, но опустить максимум, то максимум принимается равным бесконечности. Иными словами, соответствие должно быть найдено хотя бы минимальное число раз, плюс еще сколько угодно. Например, \d{7} будет соответствовать только первым семи цифрам (к примеру, местному номеру в Северной Америке или первым семи цифрам более длинного номера), в то время как \d{7,} будет соответствовать любому номеру телефона, даже международному (если только он не окажется короче семи цифр). Нет специального способа задания «не более чем» определенного числа раз. Нужно просто сказать, например, {0,5}, чтобы найти не более пяти произвольных символов.

Некоторые сочетания минимума и максимума встречаются так часто, что в Perl для них имеются особые квантификаторы. Мы уже видели +, означающий то же, что {1,}, или «хотя бы один из предшествующих элементов». Имеется также *, что равносильно {0,}, или «ноль или более предшествующих элементов», а также ?, что равносильно {0,1}, или «ноль или один предшествующий элемент» (т. е. предшествующий элемент является необязательным).

Применяя квантификаторы, следует кое в чем проявлять осторожность. Во-первых, квантификаторы Perl по умолчанию являются *жадными*. Это означает, что они ищут самое длинное соответствие шаблону из всех возможных. Например, если искать /\d+/ в строке 1234567890, то будет найдена вся строка. За этим нужно следить, особенно если используется символ «.», соответствующий любому символу. Часто встречаются строки типа:

```
larry:JYHtPh0./NJTU:100:10:Larry Wall:/home/larry:/bin/tcsh
```

в которых выполняется поиск `larry:` с помощью `/.+:/`. Однако, поскольку квантификатор + является жадным, этот шаблон включит все, вплоть до и в том числе `/home/larry:`, поскольку отыскивается самое длинное соответствие перед последним двоеточием, включая другие двоеточия. Иногда этого можно избежать, используя отрицание класса, т. е. с помощью выражения `/[^:]+:/`, которое требует найти один или несколько символов, не являющихся двоеточием (как можно больше), вплоть до первого двоеточия. Эта маленькая крышечка служит для отрицания класса символов.[1] Другой момент, который надо иметь в виду, – это стремление регулярных выражений находить возможно более *раннее* соответствие. Оно даже берет верх над жадностью. Поскольку просмотр строки происходит слева направо, это означает, что шаблон будет найден как можно левее, даже если в другом месте есть более длинное соответствие. (Регулярные выражения могут быть жадными, но не тянут с вознаграждением.) Предположим, например, что используется команда подстановки (`s///`) в строке по умолчанию, т. е. с переменной `$_`, и требуется удалить строку символов `x` из середины строки. Если сказать:

```
$_ = "fred xxxxxxx barney";
s/x*//;
```

то эффекта не будет никакого! Это связано с тем, что `x*` (означающее ноль или более символов `x`) соответствует «ничему» в начале строки, поскольку нулевая строка имеет длину ноль символов, а ясно, как божий день, что перед символом `f` в слове `fred` сидит нулевая строка.[2]

И еще об одной вещи нужно знать. По умолчанию квантификаторы применяются к единственному предшествующему символу, поэтому `/bam{2}/` найдет `bamm`, но не `bambam`. Для того чтобы применить квантификатор к нескольким символам, поставьте скобки. Поэтому для поиска `bambam` используйте шаблон `/(bam){2}/`.

Минимальное соответствие

Если при работе со старыми версиями Perl требуется, чтобы поиск не был жадным, то необходимо использовать отрицание класса. (На самом деле все равно получится жадный поиск, но несколько более сдержанный.)

В новых версиях Perl можно принудительно задать нежадный, минимальный поиск, поместив после любого квантификатора вопросительный знак. Теперь тот же наш поиск имени пользователя должен выглядеть как `/.*?:/`. Этот `.*?` будет теперь стараться найти не как можно больше символов, а как можно меньше, поэтому остановка произойдет после первого двоеточия, а не последнего.

---

[1] Мы просим извинения, но это обозначение выбрано не нами, и не стоит винить нас. Просто отрицание классов символов обычно так записывается в культуре Unix.

[2] Не отчаивайтесь. Даже авторы иногда попадают в ловушку в этих вещах.

Как поймать на слове

При попытке найти совпадение с шаблоном проверка осуществляется с каждого места, пока не будет найдено соответствие. *Привязка* (*anchor*) позволяет ограничить места, где ищется шаблон. В сущности, привязка есть нечто, соответствующее «ничему», но особому виду «ничего», зависящему от окружения. Можно назвать ее также правилом, ограничением, утверждением. Как бы она ни была названа, она пытается найти нечто нулевой длины, с успехом или без. (Неуспех просто означает, что шаблон нельзя найти этим конкретным способом. Если можно попробовать еще какие-то способы, то будут продолжены попытки найти шаблон этими способами.)

Специальный символ `\b` соответствует границе слова, которая определяется как «ничто» между символом слова (`\w`) и символом, не принадлежащим слову (`\W`), в любом порядке. (Символы, которые не существуют перед началом и после конца строки, не считаются символами слова.) Например,

```
/\bFred\b/
```

будет соответствовать `Fred` как в `The Great Fred`, так и в `Fred the Great`, но не в `Frederick the Great`, потому что за `d` в `Frederick` не следует символ слова.

В аналогичное русло укладываются символы привязки к началу или концу строки. Символ вставки (`^`), будучи первым символом строки, соответствует «ничему» в начале строки. Поэтому шаблон `/^Fred/` будет соответствовать `Fred` в `Frederick the Great`, но не в `The Great Fred`, тогда как `/Fred^/` не будет найден ни в одной из этих строк. (На самом деле в нем вообще мало смысла.) Знак доллара (`$`) работает так же, как символ вставки, но соответствует «ничему» не в начале, а в конце строки.[1]

Теперь, вероятно, читатель уже сможет определить, что говоря:

```
next LINE if $line =~ /^#/;
```

мы имеем ввиду: «перейти к следующей итерации цикла `LINE`, если данная строка начинается с символа `#`».

Ранее мы сказали, что последовательность `\d{7,11}` будет соответствовать числу длиной от семи до одиннадцати цифр. Будучи абсолютно верным, это утверждение вводит в заблуждение: задание этой последовательности в реальном операторе поиска по шаблону, `/\d{7,11}/`, не исключает того, что после одиннадцати соответствующих шаблону цифр не окажется других, ему не соответствующих! Для получения желаемого результата часто требуется осуществлять привязку шаблонов с квантификаторами с одного или обоих концов.

---

[1] Это слишком упрощенно, поскольку предполагается, что строка не содержит символа перевода строки; `^` и `$` являются, в сущности, привязками для начала и конца строчек (lines), а не строк – последовательностей символов (strings). Мы постараемся разобраться со всем этим в главе 5 «Поиск по шаблону» (насколько с этим вообще можно разобраться).

Ссылки на найденный текст

Ранее говорилось, что объединять элементы, на которые действует квантификатор, можно с помощью скобок, но можно использовать скобки и для запоминания частей найденного. Пара скобок вокруг части регулярного выражения приводит к тому, что символы, соответствующие этой части в найденном, будут запомнены и смогут использоваться в дальнейшем. При этом критерии поиска не изменяются, поэтому `/\d+/` и `/(\d+)/` будут искать самую длинную последовательность цифр, но в последнем случае они будут запомнены в специальной переменной, к которой позже можно обратиться.

Способ ссылки на найденную часть строки зависит от того, откуда это нужно сделать. Внутри того же самого регулярного выражения это делается с помощью обратной косой черты, за которой следует целое число. Целое число, соответствующее данной паре скобок, определяется путем подсчета числа левых скобок с начала шаблона, начиная с единицы. Например, для поиска чего-то похожего на тег HTML, типа `<B>Bold</B>`, можно использовать шаблон `/<(.*?)>.*?<\/\1>/`. Он требует, чтобы обе его части соответствовали в точности одинаковой строке, как `B` в данном примере.

Вне самого регулярного выражения, например, в части замены для оператора подстановки, используется `$` и целое число за ним, т. е. обычная скалярная переменная, именем которой является число. Поэтому для того чтобы переставить местами первые два слова в строке, можно использовать оператор:

```
s/(\S+)\s+(\S+)/$2 $1/
```

В правой части подстановки (между вторым и третьим символами косой черты) находится просто необычная разновидность строки в двойных кавычках, поэтому и можно производить в ней интерполяцию переменных, включая переменные с найденным текстом. Это важное понятие: интерполяция (в контролируемых условиях) – это одна из причин, по которым Perl является хорошим языком обработки текста. Другой причиной, конечно, является поиск по шаблону. Регулярные выражения предназначены для того, чтобы разобрать на кусочки, а интерполяция – чтобы собрать все вместе снова. Возможно, не все потеряно для Шалтая-Болтая.

Обработка списков

Где-то в начале этой главы мы говорили о том, что в Perl есть два основных контекста: скалярный контекст (для работы с отдельными объектами) и списочный контекст (для работы с множественными объектами). Многие из описывавшихся до сего времени традиционных операторов были по своему действию чисто скалярными. Они всегда принимают одиночные аргументы (или пары одиночных аргументов в бинарных операторах) и всегда производят одиночный результат, даже в списочном контексте. Поэтому если есть запись:

```
@array = (1 + 2, 3 - 4, 5 * 6, 7 / 8);
```

то список в правой части содержит ровно четыре значения, поскольку результатом обычных математических операторов всегда являются скалярные величины, даже в списочном контексте, возникающем при присваивании массиву.

Однако другие операторы Perl могут порождать как скалярные, так и списочные значения, в зависимости от контекста. Они просто «знают», что от них ожидается – скаляр или список. Но как можно узнать об этом? Оказывается, это нетрудно сделать, если усвоить некоторые ключевые понятия.

Во-первых, списочный контекст должен быть предоставлен чем-то, находящимся «вокруг». В предшествующем примере его предоставляет присваивание списка. Ранее мы видели, что его предоставляет оператор цикла `foreach`. Оператор `print` также предоставляет его. Но не надо заучивать это все.

Посмотрев на синтаксические сводки, разбросанные по оставшейся части книги, можно увидеть операторы, в определении которых в качестве принимаемого аргумента указан *LIST*. Это и есть операторы, предоставляющие *списочный* контекст. Всюду в этой книге *LIST* используется как специфический технический термин, означающий «синтаксическая конструкция, предоставляющая списочный контекст». Например, найдя `sort`, вы обнаружите такую синтаксическую сводку:

```
sort LIST
```

Это значит, что `sort` предоставляет списочный контекст для своих аргументов.

Во-вторых, во время компиляции (т. е. когда Perl производит синтаксический анализ программы и транслирует ее во внутренние коды операций) каждый оператор, принимающий *LIST*, предоставляет списочный контекст для каждого синтаксического элемента этого *LIST*. Поэтому каждому оператору верхнего уровня или объекту в *LIST* известно на этапе компиляции, что от него ожидается создание самого лучшего списка, на который он способен. Это означает, что если сказать:

```
sort @dudes, @chicks, other();
```

то каждый из `@dudes`, `@chicks` и `other()` знает на этапе компиляции, что от него ждут список, а не скалярную величину. Для отражения этого компилятор генерирует внутренние коды операций.

Позже, на этапе выполнения (когда внутренние коды операций интерпретируются фактически), каждый из этих элементов в *LIST* порождает свой список, а затем (это важно) все отдельные списки объединяются вместе, друг за другом, в один общий список. Этот расплющенный одномерный список и передается в конечном итоге функции, которая требовала *LIST*. Поэтому, если `@dudes` содержит `(Fred,Barney)`, `@chicks` содержит `(Wilma,Betty)`, а функция `other()` возвращает список из одного элемента `(Dino)`, то *LIST*, который получит `sort`, будет таким:

```
(Fred,Barney,Wilma,Betty,Dino)
```

а *LIST*, который возвратит sort, будет следующим:

```
(Barney,Betty,Dino,Fred,Wilma)
```

Одни операторы порождают списки (например, keys), другие потребляют их (например, print), а третьи преобразуют один список в другой (например, sort). Операторы последней категории можно рассматривать как фильтры, но в них, в отличие от оболочки, поток данных идет справа налево, поскольку списочные операторы воздействуют на аргументы, передаваемые им справа. Можно расположить подряд несколько списочных операторов:

```
print reverse sort map {lc} keys %hash;
```

В этом выражении функция keys получает ключи из хеша %hash и возвращает их функции map, которая переводит их все в нижний регистр, применяя к каждому оператор lc, и передает функции sort, которая сортирует их и передает функции reverse, которая обращает порядок элементов списка и передает их функции print, которая их выводит.

Как видите, описать это на Perl куда проще, чем на естественном языке.

Есть много других случаев, когда обработка списка порождает более естественный код. Невозможно все их здесь перечислить, но чтобы показать пример, вернемся на минуту к регулярным выражениям. Мы уже обсуждали применение шаблонов для поиска в скалярном контексте, но если использовать шаблон в списочном контексте, происходит нечто иное: все ссылки на найденный текст вытаскиваются в виде списка. Допустим, вы производите поиск в журнальном файле или почтовом ящике и хотите разобрать строку, содержащую время в формате «12:59:59 am». Можно сделать это так:

```
($hour, $min, $sec, $ampm) = /(\d+):(\d+):(\d+) *(\w+)/;
```

Это удобный способ присвоить значения одновременно нескольким переменным. Но ничуть не сложнее сказать:

```
@hmsa = /(\d+):(\d+):(\d+) *(\w+)/;
```

и поместить все четыре значения в один массив. Как ни странно, но, отделяя мощь регулярных выражений от мощи выражений Perl, списочный контекст увеличивает мощь языка. Мы редко признаемся в этом, но Perl, помимо того, что является диагональным языком, фактически является и ортогональным языком. Можно один пирог съесть дважды.

Чего вы не знаете, то вам не навредит (сильно)

В заключение позвольте еще раз вернуться к концепции Perl как естественного языка. Те, кто разговаривает на естественном языке, могут иметь разный уровень мастерства, предпочитать какие-то подмножества языка, учить

язык по ходу дела и обычно с пользой применять язык, не дожидаясь того времени, когда изучат его целиком. Вы не знаете пока Perl целиком, как не знаете целиком и тот язык, на котором разговариваете. Но в культуре Perl это официально считается нормальным. Вы можете с пользой работать на Perl, даже если мы еще не рассказали вам, как создавать собственные подпрограммы. Мы едва коснулись того, как применять Perl для системного администрирования, быстрого создания прототипов, сетевого взаимодействия или в качестве объектно-ориентированного языка. Об этих вопросах можно написать целые главы. (Если вдуматься, то мы это уже сделали.)

Но в конечном итоге читатель должен выработать собственное отношение к Perl. Как художник, вы имеете право на собственные муки творчества. Мы можем показать, как *мы* пишем картины, но не можем показать, как *вы* пишете картины. Есть Много Способов Сделать Это.

Получите причитающееся вам удовольствие.

II

Анатомия Perl

2

Всякая всячина

Мы начнем с малого и посвятим эту главу элементам Perl.

Поскольку мы начинаем с малого, продвижение по следующим главам будет вынужденно происходить от малого к большому. Это значит, что мы выбрали восходящий принцип, начав с мельчайших составляющих программ Perl и собирая из них более сложные структуры, подобно тому, как молекулы составляются из атомов. Недостаток такого подхода в том, что из-за деревьев можно не увидеть леса, потерявшись в нагромождении подробностей. Преимущество же заключается в том, что вам будут понятны примеры, приводимые по ходу изложения. (Если вы предпочитаете нисходящий порядок, переверните книгу и читайте главы в обратном порядке.)

Каждая глава базируется на материале предшествующей главы (или *последующей*, если вы читаете с конца), поэтому будьте осторожны, если у вас есть привычка перескакивать через страницы.

Разумеется, мы будем только приветствовать, если вы заглянете в справочные материалы, имеющиеся в конце книги. (Это не считается перескакиванием через страницы.) В частности, отдельные слова, выделенные моноширинным шрифтом, скорее всего можно найти в главе 29 «Функции». Мы старались сделать изложение не зависящим от операционной системы, но если вы незнакомы с терминологией Unix и столкнетесь со словом, имеющим иной смысл, чем вы себе представляете, следует поискать его в глоссарии. Если не поможет глоссарий, то, вероятно, поможет предметный указатель.

Атомы

Хотя есть много незаметных вещей, которые происходят за кулисами и о которых мы вскоре расскажем, самыми маленькими объектами, с которыми обычно приходится работать в Perl, являются отдельные символы. Мы говорим именно о символах: в прежние времена Perl легко путал байты с символами, а символы с байтами, но в наступившую эпоху глобальных сетей нужно тщательно различать эти две вещи.

Можно, конечно, писать на Perl, ориентируясь исключительно на 7-разрядный набор символов ASCII. Но на Perl можно писать и с использованием любого 8- или 16-разрядного набора символов, будь это национальный или какой-либо традиционный алфавит. Однако, если вы станете применять какой-либо из старых (не-ASCII) наборов символов, помните, что символы не-ASCII можно использовать только в строковых литералах. Необходимо обеспечить соответствие семантики программы выбранному национальному алфавиту. Так, работая с 16-разрядным набором азиатских национальных символов не забывайте, что Perl обычно воспринимает каждый из этих символов как два байта, а не один символ.

В главе 15 «Unicode» сказано о том, что недавно в Perl была добавлена кодировка Unicode.[1] Эта поддержка относится к языку в целом: можно использовать символы Unicode как в идентификаторах (например, именах переменных), так и в строках литералов. В случае применения Unicode не нужно беспокоиться о том, сколько бит или байт занимает символ. Perl представляет дело так, как если бы все символы Unicode имели одинаковый размер (а именно 1), даже если какой-то символ имеет внутреннее представление из нескольких байт. Обычно Perl использует для внутреннего представления кодировку с переменной длиной UTF-8. (Например, «смайлик» в кодировке Unicode, U-263A, внутренне представляется последовательностью из трех символов.)

Если вы позволите несколько развить нашу аналогию с физическими элементами, то символы атомарны в том же самом смысле, что и атомы различных элементов. Да, они состоят из более мелких частиц, которые называются битами и байтами, но если расколоть символ на части (очевидно, в символьном ускорителе), то отдельные биты и байты утратят отличительные «химические» свойства целого символа. Как нейтроны являются деталями реализации атома урана-238, так и байты являются деталями реализации символа U-263A.

Поэтому мы будем стремиться говорить: «символы», имея в виду символы, и говорить: «байты», имея в виду байты. Но мы не хотели никого напугать: вы по-прежнему можете легко по старинке обрабатывать байты. Надо только сообщить Perl, что вы по-прежнему хотите рассматривать байты

---

[1] Несмотря на наш восторг по поводу поддержки Unicode, большинство наших примеров написано на ASCII, поскольку не у всех еще имеется приличный редактор Unicode.

как символы. Это можно сделать с помощью прагмы (директивы) `use bytes` (см. главу 31 «Модули прагм»). Даже если вы этого не сделаете, Perl все равно позаботится о том, чтобы хранить маленькие символы в 8 разрядах, как вам того и хотелось бы.

А теперь хватит о мелочах. Переходим к более крупным и приятным вещам.

Молекулы

Perl является языком *свободной формы* (*free-form language*), но это не значит, что Perl вообще бесформенный. Компьютерщики обычно используют этот термин для обозначения языка, в котором пробелы, символы табуляции и перевода строки можно помещать где угодно – за исключением тех мест, где этого делать нельзя.

Первое очевидное место, в котором нельзя ставить пробельные символы, – это середина лексемы. *Лексема* (*token*) – это последовательность символов с единицей смыслового содержания, примерно как слово в естественном языке. Но, в отличие от обычных слов, лексема может содержать символы, не являющиеся буквами, надо лишь, чтобы они сплоченно составляли смысловую единицу. (В этом отношении они более похожи на молекулы, которые не обязательно состоят из атомов только одного вида.) Например, числа и математические операторы считаются лексемами. *Идентификатор* (*identifier*) – это лексема, которая начинается с буквы или символа подчеркивания и содержит только буквы, цифры и символы подчеркивания. Лексема не может содержать пробельных символов, поскольку в результате она оказалась бы разбитой на две лексемы, так же как пробел в слове английского языка превращает его в два слова.[1]

Пробельный символ допускается помещать между любыми двумя лексемами, но *обязательным* его присутствие является только между такими лексемами, которые без него могут быть приняты за одну. Для этой цели могут использоваться любые символы из числа пробельных. Символы перевода строки отличаются от пробелов и символов табуляции только внутри строк, заключенных в кавычки, форматов и некоторых строчно-ориентированных типов заключения в кавычки. Конкретно символ новой строки не служит концом команды (утверждения), как это имеет место в некоторых других языках (например, FORTRAN или Python). В Perl команды завершаются точкой с запятой, как в C и производных от него языках.

Пробельные символы Unicode допускается использовать в Unicode-программах Perl, но при этом следует проявлять осторожность. Применяя особые разделители абзацев и строк из Unicode, учтите, что Perl может пронумеровать строки не так, как это сделает ваш текстовый редактор, из-за чего

[1] Проницательный читатель отметит, что литеральные строки могут содержать пробельные символы. Но в строках это возможно только потому, что на концах их стоят двойные кавычки, удерживающие пробелы на месте.

может быть затруднено толкование сообщений об ошибках. Лучше придерживаться использования старых добрых символов перевода строки.

Лексемы распознаются путем жадного поиска: если в некотором месте анализатор Perl имеет выбор между короткой и длинной лексемами, он предпочтет длинную. Если вам нужно, чтобы это были две лексемы, вставьте между ними пробельный символ. (Мы в любом случае стараемся ставить лишние пробелы по краям большей части операторов, чтобы повысить удобочитаемость.)

Комментарии задаются символом # и простираются от него и до конца строки. Комментарий действует как пробельный символ в отношении разделения лексем. В языке Perl содержимое комментария не имеет никакого особого смысла, что бы в нем ни находилось.[1]

Еще одна особенность связана с тем, что если строка начинается с = и в этом месте допустимо поместить команду, то Perl игнорирует все, от этой строки вплоть до строки, начинающейся с `=cut`. Игнорируемый текст считается документацией, или *pod* («plain old documentation»). В дистрибутив Perl входят программы, извлекающие комментарии типа pod из модулей Perl и преобразующие их в простой текст, страницы руководства, документы LaTeX, HTML, а в скором будущем, возможно, XML. Анализатор Perl, напротив, извлекает код Perl из модулей Perl и игнорирует pod. Можно рассматривать это как второй способ создания многострочных комментариев. Можете не верить, но для модулей Perl, документированных таким способом, документация никогда не теряется. О деталях документирования программ рассказывается в главе 26 «Документирование программ»; там же описывается обработка многострочных комментариев средствами Perl.

Но не пренебрегайте и обычным символом комментария. Есть что-то успокаивающее в аккуратной колонке символов # с левой стороны многострочного комментария. Она немедленно говорит вашим глазам: «Это не код». Заметьте, что даже в языках, имеющих средства создания многострочных комментариев, таких как C, программисты часто все равно помещают колонку символов * по левому краю комментариев. Внешний вид часто важнее, чем может показаться.

В Perl, как в химии и в языке, можно строить все более и более крупные структуры из мелких. Мы уже употребляли слово *команда* (*statement*): это просто последовательность лексем, составляющих вместе предложение в повелительном наклонении. Последовательность команд можно объединить в *блок*, который заключается в фигурные скобки (braces), любовно называемые также «завитушками» теми, кто считает, что braces – это подтяжки.[2]

---

[1] На самом деле это небольшая ложь. Анализатор Perl все же ищет ключи командной строки в начальной строке `#!` (см. главу 19 «Интерфейс командной строки»). Он может также интерпретировать директивы номера строки, создаваемые некоторыми препроцессорами (см. раздел «Генерация Perl из других языков» главы 24 «Распространенные приемы программирования»).

[2] Другое значение слова braces. – *Примеч. перев.*

Блоки, в свою очередь, могут объединяться в еще более крупные блоки. Некоторые блоки действуют как *подпрограммы* (*subroutines*), которые могут объединяться в *модули* (*modules*), которые могут объединяться в *программы* (*programs*). Но мы слишком забегаем вперед – это темы последующих глав. Лучше построим из символов еще несколько лексем.

Встроенные типы данных

Прежде чем начать разговор о различных типах лексем, которые можно построить из символов, требуется ввести еще несколько абстракций. Точнее, нам нужны три типа данных.

Языки программирования различаются по числу и видам имеющихся в них типов данных. В отличие от некоторых широко распространенных языков, в которых есть много запутанных типов для сходных между собой видов значений, в Perl довольно мало встроенных типов данных. Возьмите C, в котором можно встретить `char`, `short`, `int`, `long`, `long long`, `bool`, `wchar_t`, `size_t`, `off_t`, `regex_t`, `uid_t`, `u_longlong_t`, `pthread_key_t`, `fp_exception_field_type` и т. д. И это только некоторые целочисленные типы! А кроме того есть числа с плавающей запятой, а также указатели, а также строки.

Всем этим сложным типам данных соответствует в Perl единственный тип: скаляр. (Обычно ничего не нужно, кроме простых типов данных Perl, но если что-то потребуется, можно определять необычные динамические типы с использованием объектно-ориентированных возможностей Perl – см. главу 12 «Объекты».) Основными тремя типами данных Perl являются *скаляры*, *массивы* скаляров и *хеши* скаляров (называемые также *ассоциативными массивами*). Некоторые предпочитают называть их не типами, а *структурами данных*. Это их право.

Скаляры являются базовым типом, на основе которого строятся более сложные структуры. Скаляр содержит одно простое значение, обычно строку или число. Элементы этого простого типа могут объединяться в один из двух составных типов. *Массив* – это упорядоченный список скаляров, обращение к которым происходит с помощью целочисленного индекса (subscript). В Perl индексация начинается с 0. Однако, в отличие от многих языков программирования, Perl считает допустимыми отрицательные индексы: вместо отсчета с начала отрицательные индексы ведут обратный отсчет с конца чего бы то ни было индексируемого. (Это относится как к обычному индексированию, так и к различным операциям с подстроками и подсписками.) Напротив, *хеш* является неупорядоченным списком пар *ключ/значение*, обращение к которым производится путем использования строки (*ключа*) в качестве индекса для поиска скаляра (*значения*), соответствующего данному ключу. Переменные всегда имеют один из этих трех типов. (Кроме переменных в Perl есть другие абстракции, которые можно рассматривать как типы данных, например, дескрипторы файлов, дескрипторы каталогов, форматы, подпрограммы, таблицы имен и записи таблиц имен.)

Абстракции – это чудесно, и по ходу дела у нас их наберется много, но в некотором отношении они бесполезны. Непосредственно с абстракцией делать ничего нельзя. Поэтому в языках программирования имеется синтаксис. Мы должны познакомить вас с некоторыми видами синтаксических термов, которые можно использовать для ввода абстрактных данных в выражения. Нам нравится технический термин *терм* при необходимости поговорить в терминах синтаксических единиц. (Однако это может создать терминальную путаницу. Вспомните просто, что школьный учитель математики говорил о термах (членах) в уравнении, и вы не сильно ошибетесь.)

Подобно членам математического уравнения, назначением термов в Perl является создание величин, с которыми действуют операторы типа сложения и умножения. Однако, в отличие от математического уравнения, Perl должен что-то *делать* с величинами, которые он вычисляет, а не просто размышлять с карандашом в руке, равны ли между собой две части уравнения. Действие чаще всего состоит в том, чтобы где-то запомнить значение:

```
$x = $y;
```

Это пример оператора *присвоения* (а не оператора равенства чисел, который в Perl записывается как ==). Присвоение получает значение из $y и помещает его в $x. Обратите внимание, что терм $x используется не ради его значения, а ради его адреса. (Прежнее значение $x будет уничтожено присвоением.) Мы говорим, что $x является *l-значением* (от *left value* – левое значение), подразумевая под этим, что данный тип памяти может присутствовать в левой части присвоения. Мы говорим, что $y является *r-значением* (от *right value* – правое значение), поскольку оно расположено в правой части.

Есть и третий тип значений, называемый *временными* значениями, который нужно понимать, если вы действительно хотите узнать, что в действительности делает Perl с вашими l-значением и r-значением. Если мы действительно выполняем какие-то математические действия и говорим:

```
$x = $y + 1;
```

то Perl берет левое значение $y и прибавляет к нему правое значение 1, в результате чего создается временное значение, которое в конечном итоге присваивается левому значению $x. Возможно, вам будет легче представить себе зрительно, что происходит, если мы сообщим, что Perl запоминает эти временные значения во внутренней структуре, называемой *стеком*.[1] Термы выражения (о которых мы говорим в данной главе) стремятся протолкнуть значения в стек, тогда как операторы выражения (о которых мы будем говорить в следующей главе) стремятся вытолкнуть их обратно, возможно, оставляя при этом в стеке временный результат, с которым будет работать сле-

[1] Стек работает способом, напоминающим подпружиненные устройства раздачи тарелок в буфетах – можно проталкивать (*push*) тарелки на верх стека (стопки), а можно выталкивать их (*pop*) обратно (если воспользоваться профессиональным жаргоном вычислительных наук).

дующий оператор. Все эти проталкивания и выталкивания обратно уравновешивают друг друга: к тому времени, когда вычисление выражения заканчивается, стек становится совершенно пуст (или столь же пуст, как в начале вычисления). Мы еще скажем ниже о временных значениях.

Некоторые термы могут быть только правыми значениями, как, скажем, `1` в примере выше, тогда как другие могут быть и левыми, и правыми значениями. В частности, как показывают вышеприведенные присвоения, переменная может выполнять ту и другую роль. Об этом и рассказывается в нашем следующем разделе.

Переменные

Неудивительно, что есть три типа переменных, соответствующие трем указанным выше абстрактным типам данных. Каждой из таких переменных предшествует так называемый *разыменовывающий префикс* (*funny character*).[1] Имя скалярной переменной всегда начинается символом `$`, даже если ссылка производится на скаляр, являющийся частью массива или хеша. Его действие несколько напоминает использование определенного артикля «the» в английском языке. Таким образом, мы имеем:

Конструкция	Значение
`$days`	Простое скалярное значение `$days`
`$days[28]`	29-й элемент массива `@days`
`$days{'Feb'}`	Значение «`Feb`» из хеша `%days`

Обратите внимание, что одно и то же имя можно использовать для `$days`, `@days` и `%days`, и при этом Perl не запутается.

Есть и другие, необычные скалярные термы, применяемые в особых ситуациях, в которые мы пока вникать не будем. Они выглядят так:

Конструкция	Значение
`${days}`	То же, что `$days`, но устраняется неоднозначность перед буквенно-цифровыми символами
`$Dog::days`	Еще одна переменная `$days`, в пакете `Dog`
`$#days`	Последний индекс массива `@days`
`$days->[28]`	29-й элемент массива, на который указывает ссылка `$days`

[1] Это еще один технический термин в вычислительных науках (если раньше он им не был, то сейчас является).

(*продолжение*)

Конструкция	Значение
`$days[0][2]`	Многомерный массив
`$days{2000}{'Feb'}`	Многомерный хеш
`$days{2000,'Feb'}`	Эмуляция многомерного хеша

Массивы в целом (или *срезы* массивов и хешей) именуются с помощью разыменовывающего символа `@`, действующего во многом аналогично словам «эти» или «те»:

Конструкция	Значение
`@days`	Массив, содержащий (`$days[0], $days[1],... $days[n]`)
`@days[3, 4, 5]`	Срез массива, содержащий (`$days[3], $days[4], $days[5]`)
`@days[3..5]`	Срез массива, содержащий (`$days[3], $days[4], $days[5]`)
`@days{'Jan','Feb'}`	Срез хеша, содержащий (`$days{'Jan'},$days{'Feb'}`)

Хеш в целом именуется с помощью `%`:

Конструкция	Значение
`%days`	`(Jan => 31, Feb => $leap ? 29 : 28, ...)`

Все эти конструкции могут также служить в качестве l-значений, задавая адрес, по которому можно присвоить значение. Для массивов, хешей и срезов массивов и хешей l-значение предоставляет местоположение для присвоения, поэтому можно одновременно присваивать им несколько значений:

```
@days = 1 .. 7;
```

Имена

Мы говорили о хранении значений в переменных, но и сами переменные (их имена и связанные с ними определения) тоже должны где-то храниться. Теоретически такие места называются *пространствами имен* (*namespaces*). В Perl есть два вида пространств имен, которые часто называются *таблицами* имен (*symbol tables*) и *областями лексической видимости* (*lexical scopes*).[1] Можно иметь произвольное количество таблиц имен и областей лексической видимости, и каждое определяемое вами имя хранится в той или иной из них. Далее мы рассмотрим оба типа пространств имен, но пока только

[1] В конкретных реализациях Perl они также называются *пакетами* (*packages*) и *панелями* (*pads*), но приведенные нами более длинные названия являются общими терминами в данной области, поэтому мы стараемся придерживаться их. Извините.

скажем, что таблицы имен являются глобальными хешами, содержащими записи таблицы имен для глобальных переменных (в том числе хешей для других таблиц имен). Напротив, лексические области видимости являются *анонимной* временной памятью, находящейся не в какой-либо таблице имен, а прикрепленной к блоку кода вашей программы. Они содержат переменные, которые можно видеть только в данном блоке. (Это мы и подразумеваем под *областью видимости*. Часть *лексическая* просто означает, что она относится к тексту, и это совсем не то, что подразумевалось бы лексикографом. Не вините нас.)

В каждом данном пространстве имен (глобальном или лексическом) для каждого типа переменных имеется собственное подпространство имен, определяемое разыменовывающим префиксом. Можно, не опасаясь конфликта, использовать одно и то же имя для скалярной переменной, массива или хеша (а также дескриптора файла, подпрограммы, метки или любимой ручной ламы). Это значит, что `$foo` и `@foo` суть две различные переменные. Согласно предшествующим правилам, это также значит, что `$foo[1]` является элементом `@foo`, никак не связанным со скалярной переменной `$foo`. Это может показаться странным, и правильно, поскольку это *действительно* странно.

Имена подпрограмм могут начинаться с символа `&`, хотя при вызове подпрограмм задание разыменовывающего префикса не обязательно. Обычно подпрограммы не считаются l-значениями, но в последних версиях Perl разрешается возвращать из подпрограммы l-значение и выполнять присвоение по отношению к нему, что выглядит как присвоение подпрограмме.

Иногда требуется имя, которым можно назвать «все, что названо foo», независимо от разыменовывающего символа. Поэтому имена записей таблицы имен могут начинаться символом *, при этом звездочка заменяет все остальные разыменовывающие префиксы. Эти записи называются *записями глобальной таблицы имен* (*typeglobs*), и есть несколько способов их использования. Они могут также выступать в качестве l-значений. Посредством присвоения типу данных *typeglob* в Perl реализуется импорт имен из одной таблицы имен в другую. Подробнее об этом тоже дальше.

Как и в большинстве языков программирования, в Perl имеется список зарезервированных слов, которые распознаются им как особые ключевые слова. Однако поскольку имена переменных всегда начинаются с разыменовывающего префикса, то зарезервированные слова фактически не конфликтуют с именами переменных. Но некоторые другие типы имен не имеют разыменовывающих префиксов, например дескрипторы файлов и метки. Для таких имен нужно беспокоиться (но не сильно) об отсутствии конфликта с зарезервированными словами. Поскольку в большинстве зарезервированных слов применяется только нижний регистр, рекомендуем выбирать имена меток и дескрипторов файлов так, чтобы они содержали заглавные буквы. Например, говоря `open(LOG, logfile)`, а не предосудительное `open(log, "logfile")`, нельзя ввести Perl в заблуждение, будто имеется в виду встроенный оператор `log` (который, конечно же, осуществляет логарифмирование). Использование дескрипторов файлов в верхнем регистре также повышает удобочитае-

мость[1] и предотвращает конфликты с зарезервированными словами, которые могут появиться в будущем. По сходным причинам модули, определяемые пользователем, обычно имеют названия, начинающиеся с заглавных букв, чтобы их было легче отличать от встроенных модулей, которые называются прагмами (директивами) и именуются только в нижнем регистре. А когда мы доберемся до объектно-ориентированного программирования, то вы увидите, что по той же причине в именах классов обычно используются заглавные буквы.

Как можно было заключить из предыдущего абзаца, в идентификаторах имеет значение регистр – `FOO`, `Foo` и `foo` являются в Perl разными именами. Идентификаторы начинаются с символа подчеркивания или буквы, могут иметь любую длину (где любая длина находится в диапазоне от 1 до 251 включительно) и могут содержать буквы, цифры и символы подчеркивания, в том числе буквы и цифры Unicode. Идеограммы Unicode тоже сойдут за буквы, но не советуем их использовать, если вы не умеете произносить их. См. главу 15 «Unicode».

Имена с разыменовывающими префиксами не обязаны, строго говоря, быть идентификаторами. Они могут начинаться с цифр, и тогда должны содержать только цифры, например `$123`. Имена, начинающиеся с символа, отличного от буквы, цифры или подчеркивания, ограничены (обычно) одним символом (например, `$?` или `$$`) и обычно имеют в Perl предопределенное значение. Например, как и в оболочке Борна, `$$` является идентификатором текущего процесса, а `$?` – статусом завершения последнего дочернего процесса.

Начиная с версии 5.6 в Perl появился открытый синтаксис для внутренних имен переменных. Всякая переменная вида `${^NAME}` является специальной переменной, зарезервированной для применения в Perl. Все эти имена, не являющиеся идентификаторами, помещаются в главную таблицу имен. Некоторые примеры можно найти в главе 28 «Специальные имена».

Можно ошибочно придти к выводу, что имена и идентификаторы – это одно и то же, однако, говоря *имя*, мы обычно подразумеваем полностью квалифицированное имя, т. е. имя, говорящее о том, из какой таблицы имен оно взято. Такое имя может быть образовано последовательностью идентификаторов, разделяемых лексемой `::`:

```
$Santa::Helper::Reindeer::Rudolph::nose
```

Это похоже на каталоги и имена файлов в именах маршрутов:

```
/Santa/Helper/Reindeer/Rudolph/nose
```

В версии этого обозначения, используемой Perl, все ведущие идентификаторы являются именами вложенных таблиц имен, а последний – именем пере-

[1] Один из принципов архитектуры Perl заключается в том, что разные вещи должны и выглядеть по-разному. Сравните это с языками, в которых делается попытка заставить разные вещи выглядеть одинаково в ущерб удобочитаемости.

менной в самой глубоко вложенной таблице имен. Например, в приведенной ранее переменной именем таблицы является `Santa::Helper::Reindeer::Rudolph::`, а фактическая переменная в этой таблице имен – `$nose` (нос). (А значением этой переменной является, конечно, «красный».)

Таблицу имен в Perl называют также *пакетом* (*package*), поэтому соответствующие переменные часто называются пакетными. Пакетные переменные номинально принадлежат пакету, в котором существуют, но являются глобальными в том смысле, что сами пакеты глобальны. Это значит, что каждый может указать имя пакета, чтобы добраться до переменной, но просто по ошибке это сделать трудно. Например, всякая программа, в которой упоминается `$Dog::bert`, запрашивает переменную `$bert` из пакета `Dog::`. А эта переменная совсем не такая, как `$Cat::bert`. См. главу 10 «Пакеты».

Переменные, относящиеся к лексической области, не принадлежат никакому пакету, поэтому имена переменных с лексической областью видимости не могут содержать последовательность "`::`". (Переменные с лексической областью видимости объявляются с помощью ключевого слова `my`.)

Поиск имен

Итак, вопрос: что в имени тебе моем? Как Perl определяет, что вы имеете в виду, говоря просто `$bert`? Хороший вопрос. Вот правила, которыми руководствуется анализатор Perl, пытаясь понять неквалифицированное имя в контексте:

1. Сначала Perl просматривает ближайший охватывающий блок и проверяет, нет ли в этом блоке данной переменной в объявлениях `my` или `our` (почитайте о них в главе 29 «Функции» и разделе «Объявления с областью видимости» главы 4 «Команды и объявления»). Если объявление `my` есть, то переменная имеет лексическую область видимости и не существует ни в каком пакете – она существует только в этой лексической области видимости (т. е. во временной памяти этого блока). Поскольку лексические области видимости безымянны, никто вне этого фрагмента не может даже назвать вашу переменную.[1]

2. Если это не удается, Perl ищет блок, охватывающий данный, и переменную с лексической областью видимости в этом более широком блоке. Если это удается, значит, переменная принадлежит только лексической

[1] При использовании объявления `our` вместо объявления `my` объявляется псевдоним с лексической областью видимости для пакетной переменной, а не действительная переменная с лексической областью видимости, как происходит в случае объявления `my`. Сторонний код может иметь доступ к настоящей переменной через ее пакет, но во всех остальных отношениях объявление `our` ведет себя как объявление `my`. Это удобно, если вы пытаетесь ограничить свое собственное использование глобальных переменных с помощью прагмы `strict` (см. описание `strict` в главе 31 «Модули прагм»). Но всегда предпочтительнее использовать `my`, если вам не нужна глобальная переменная.

области видимости от точки объявления до конца блока, в котором она объявлена, включая вложенные блоки, например тот, из которого мы пришли на шаге 1. Если Perl не находит объявления, он повторяет шаг 2, пока не кончатся охватывающие блоки.

3. Когда Perl не может больше найти блоки, охватывающие данные, он рассматривает всю компилируемую единицу и ищет в ней объявления, как если бы это был блок (*компилируемой единицей* является весь текущий файл или строка, компилируемая в данный момент оператором `eval STRING`). Если компилируемой единицей является файл, то это самая большая возможная лексическая область видимости, и Perl прекращает дальнейший поиск лексических областей видимости, поэтому мы переходим к шагу 4. Если компилируемой единицей является строка, то все не так просто. Строка, компилируемая как код Perl на этапе выполнения, представляется блоком в лексической области видимости, из которой вызван `eval STRING`, хотя действительными границами лексической области видимости являются не фигурные скобки, а границы строки, содержащей код. Поэтому, если Perl не находит переменной в лексической области видимости строки, действие происходит так, как если бы `eval STRING` был блоком, и выполняется возврат к шагу 2, только на этот раз поиск начинается с лексической области видимости оператора `eval STRING`, а не с области внутри его строки.
4. Если мы попали сюда, это значит, что Perl не нашел для переменной никаких объявлений (ни `my`, ни `our`). Теперь Perl отказывается от поиска переменной с лексической областью видимости и считает, что переменная является пакетной. Если действует директива `strict`, то в данном случае будет получена ошибка, если только эта переменная не является одной из предопределенных переменных Perl или не импортирована в текущий пакет. Дело в том, что данная директива запрещает использование неквалифицированных глобальных имен. Однако мы еще не покончили с лексическими областями видимости. Perl осуществляет такой же поиск лексических областей видимости, как на шагах с 1 по 3, но на этот раз ищет не объявления переменных, а объявления пакетов. Если он находит такое объявление пакета, то полагает, что текущий код компилируется для указанного пакета, и делает имя объявленного пакета префиксом переменной.
5. Не обнаружив объявления пакета в окружающей лексической области видимости, Perl ищет имя переменной в непоименованном пакете верхнего уровня, именем которого оказывается `main`, если только у него не отсутствует тег имени. Поэтому, в отсутствие противоречащих тому объявлений, `$bert` означает то же, что `$::bert`, что означает то же, что `$main::bert`. (Но поскольку `main` является пакетом в непоименованном пакете верхнего уровня, то это также `$::main::bert`, и `$main::main::bert`, `$::main::main::bert` и т. д. Это можно рассматривать как бесполезный факт. Однако загляните в раздел «Таблицы имен» главы 10 «Пакеты».)

Есть несколько следствий из этих правил поиска, которые могут быть неочевидными, поэтому сформулируем их явно.

- Поскольку файл является наибольшей возможной лексической областью видимости, переменную с лексической областью видимости нельзя увидеть вне файла, в котором она определена. Файловые области видимости не могут быть вложенными.
- Любой фрагмент Perl компилируется, по крайней мере, в одной лексической области видимости и ровно в одной области видимости пакета. Обязательной лексической областью видимости является, конечно, сам файл. Дополнительные лексические области видимости образуют все охватывающие блоки. Весь код Perl компилируется также в области видимости ровно одного пакета, и хотя объявление любого пакета имеет лексическую область видимости, сами пакеты не ограничены лексически, т. е. являются глобальными.
- Поиск неквалифицированного имени переменной может, следовательно, производиться во многих лексических областях видимости, но только в одной области видимости пакета – той, которая активна в данный момент (что лексически определено).
- Имя переменной может быть связано только с одной областью видимости. Хотя в любом месте программы действуют, по крайней мере, две разные области видимости (лексическая и пакета), переменная может существовать только в одной из этих областей видимости.
- Неквалифицированное имя переменной может, следовательно, быть разрешено только с одним адресом памяти: либо в первой охватывающей лексической области видимости, в которой оно объявлено, либо в текущем пакете – но не в обоих местах одновременно. Поиск прекращается, как только разрешается этот адрес памяти, а все адреса памяти, которые могли быть найдены при продолжении поиска, оказываются в результате скрытыми.
- Местонахождение обычного имени переменной можно полностью определить на этапе компиляции.

Теперь вам все известно о том, как компилятор Perl работает с именами, но иногда возникает проблема, связанная с тем, что на этапе компиляции нужное имя *не известно*. Иногда требуется обратиться к чему-либо косвенным образом; мы называем это задачей *косвенности*. Perl предоставляет для этого следующий механизм: буквенно-цифровое имя переменной можно заменить блоком, содержащим выражение, которое возвращает *ссылку* на реальные данные. Например, вместо:

```
$bert
```

можно сказать:

```
${ some_expression() }
```

и если функция `some_expression()` возвращает ссылку на переменную `$bert` (или даже строку `bert`), то это равносильно тому, что вы сказали бы `$bert`. С другой стороны, если функция возвращает ссылку на `$ernie`, то вы получите

его переменную. Продемонстрированный синтаксис является наиболее общим (и наименее доходчивым) вариантом косвенности, но мы расскажем о некоторых его удобных разновидностях в главе 8 «Ссылки».

Скалярные значения

Скаляр, будь он поименован прямо или косвенно, находись он в переменной, элементе массива или временной величине, всегда содержит единственное значение. Этим значением может быть число, строка или ссылка на другой фрагмент данных. Значение может также вовсе отсутствовать, и тогда скаляр называют *неопределенным* (*undefined*). Хотя можно говорить, что скаляр «содержит» число или строку, скаляры не имеют типа: не требуется объявлять, что скаляр имеет тип целого числа или числа с плавающей точкой, строки или какой-либо еще.[1]

Perl хранит строки как последовательности символов без каких-либо ограничений на размер или содержимое. Простыми словами, не нужно решать заранее, насколько длинными будут строки, а включать в них можно любые символы, в том числе нулевые байты. Perl старается сохранять числа как целые со знаком, а если это невозможно, то как величины с плавающей точкой и двойной точностью в родном формате машины. Точность значений с плавающей точкой не бесконечна. Об этом нужно помнить, иначе равенства типа `(10/3 == 1/3*10)` по таинственным причинам не будут выполняться.

При необходимости Perl осуществляет преобразование между различными подтипами, поэтому можно обращаться с числом как со строкой и со строкой как с числом, и Perl будет поступать как должно. Чтобы преобразовать строку в число, Perl внутренне использует нечто вроде функции *atof*(3) из библиотеки C. Чтобы преобразовать число в строку, он на большинстве машин осуществляет действие, эквивалентное вызову *sprintf*(3) с форматом `"%.14g"`. Попытки преобразования нечисловой строки типа `foo` в число дают в результате числовой 0 и вызывают вывод предупредительных сообщений, если он включен. Примеры определения типа данных, содержащихся в строке, можно найти в главе 5 «Поиск по шаблону».

Хотя строки и числа почти всегда взаимозаменяемы, со ссылками дело обстоит несколько иначе. Ссылки являются строго типизованными, неприводимыми указателями со встроенным счетчиком обращений и вызовом деструктора. Это означает, что их можно применять в случае создания сложных ти-

---

[1] В будущих версиях Perl будет разрешено вводить объявления типов `int`, `num` и `str`, но не для того, чтобы осуществлять строгую типизацию, а для помощи оптимизатору в вопросах, которые он не смог бы выяснить самостоятельно. Обычно это нужно только в отлаженном коде, который должен выполняться очень быстро, поэтому мы не будем пока рассказывать о том, как это можно сделать. Необязательные типы используются также механизмом псевдохеширования, и в этом случае они могут действовать так, как типы в более сильно типизированном языке. Дополнительные сведения см. в главе 8 «Ссылки».

пов данных, в том числе определяемых пользователем объектов. При этом они все же остаются скалярами, потому что какой бы сложной ни была структура данных, иногда требуется рассматривать ее как одиночное значение.

Под *неприводимостью* мы понимаем то, что нельзя, например, преобразовать ссылку на массив в ссылку на хеш. Ссылки не преобразуются в другие типы указателей. Однако если использовать ссылку как число или строку, то получается числовое или строковое значение, которому обеспечено сохранение уникальности ссылки, даже несмотря на то, что «ссылочность» значения утрачивается при его копировании из действительной ссылки. Такие величины можно сравнивать или определять их тип. Но сделать что-либо еще с этими величинами трудно, поскольку нельзя преобразовать числа или строки обратно в ссылки. Обычно это не вызывает проблем, поскольку Perl не принуждает вас заниматься арифметикой указателей – и даже не разрешает ее. Дополнительные сведения о ссылках см. в главе 8 «Ссылки».

Числовые литералы

Числовые литералы задаются в одном из нескольких обычных[1] форматов чисел с плавающей запятой или целых чисел:

```
$x = 12345;              # целое
$x = 12345.67;           # с плавающей точкой
$x = 6.02e23;            # научная нотация
$x = 4_294_967_296;      # подчеркивания для лучшей читаемости
$x = 0377;               # восьмеричное
$x = 0xffff;             # шестнадцатеричное
$x = 0b1100_0000;        # двоичное
```

Поскольку Perl использует запятую в качестве разделителя в списках, она не может выступать в качестве разделителя тысяч в больших числах. Для этой цели Perl разрешает применять символ подчеркивания. Символ подчеркивания действует только в числовых литералах, определяемых в вашей программе, но не в строках, выступающих как числа, или данных, считываемых из какого-либо другого места. Аналогичным образом префиксы `0x` для шестнадцатеричных чисел, `0b` для двоичных и `0` для восьмеричных действуют только в литералах. Автоматическое преобразование строки в число не распознает эти префиксы: необходимо выполнять явное преобразование[2] с помощью функции `oct`, которая обрабатывает также и шестнадцатеричные и двоичные числа с префиксами `0x` или `0b`.

---

[1] В смысле обычных в культуре Unix. Если вы из другой культуры, добро пожаловать в нашу!

[2] Иногда программистам кажется, что Perl должен вместо них преобразовывать все входные данные. Но в окружающей жизни слишком много десятичных чисел с ведущими нулями, для того чтобы Perl делал это автоматически. Например, почтовый индекс офиса O'Reilly & Associates в Кембридже, шт. Массачусетс выражается числом `02140`. Почтовое ведомство оказалось бы в замешательстве, если бы ваша программа печати почтовых наклеек преобразовала `02140` в десятичное `1120`.

Строковые литералы

Строковые литералы обычно заключаются в одинарные или двойные кавычки. Они действуют во многом так же, как в оболочке Unix: строковые литералы в двойных кавычках подвергаются интерполяции обратного слэша и переменных, а строки в одинарных кавычках – нет (за исключением `\'` и `\\`, чтобы можно было включать одинарные кавычки и обратную косую черту в строки, заключенные в одинарные кавычки). Если требуется встроить другую последовательность с обратного слэша, например `\n` (перевод строки), то следует выбрать формат в двойных кавычках. (Последовательности с обратного слэша называются также *escape-последовательностями*, поскольку позволяют временно «избежать» обычной интерпретации символов.)

Строка, заключаемая в одинарные кавычки, должна отделяться пробелом от предшествующего слова, поскольку одинарная кавычка является допустимым, хотя и архаичным, символом идентификатора. Вместо него теперь используется более отчетливо видимая последовательность `::`. Это означает, что `$main'var` и `$main::var` суть одно и то же, но считается, что вторую запись легче читать и человеку, и программе.

Строки, заключенные в двойные кавычки, подвергаются различным видам интерполяции символов, многие из которых окажутся знакомы программирующим на других языках. Они перечислены в табл. 2.1.

Таблица 2.1. Escape-последовательности с обратным слэшем

Код	Значение
`\n`	Перевод строки (обычно LF)
`\r`	Возврат каретки (обычно CR)
`\t`	Горизонтальная табуляция
`\f`	Подача листа
`\b`	Забой
`\a`	Тревога (звонок)
`\e`	ESC-символ
`\033`	ESC в восьмеричном виде
`\x7f`	DEL в шестнадцатеричном виде
`\cC`	<Ctrl>+<C>
`\x{263a}`	Unicode (смайлик)
`\N{NAME}`	Именованный символ

Обозначение `\N{NAME}` используется только совместно с прагмой `charnames`, описанной в главе 31 «Модули прагм». Она позволяет задавать символические имена символов, например, `\N{GREEK SMALL LETTER SIGMA}`, `\N{greek:Sigma}` или `\N{sigma}` – в зависимости от способа вызова директивы. См. также главу 15 «Unicode».

Существуют также escape-последовательности для изменения регистра или «метасвойств» последующих символов, приведенных в табл. 2.2.

Таблица 2.2. Escape-последовательности трансляции

Код	Значение
`\u`	Перевести следующий символ в верхний регистр («регистр заголовка» (titlecase) в Unicode)
`\l`	Перевести следующий символ в нижний регистр
`\U`	Перевести все следующие символы в верхний регистр
`\L`	Перевести все следующие символы в нижний регистр
`\Q`	Поставить обратную косую черту во всех последующих символах, не являющихся буквой или цифрой
`\E`	Прекратить действие `\U` `\L` или `\Q`

Символы перевода строки можно непосредственно вставлять в строки, т. е. они могут начинаться и завершаться на разных строчках. Часто это бывает удобно, но означает также, что если вы забудете поставить закрывающую кавычку, то сообщение об ошибке последует только тогда, когда Perl найдет очередную строку, содержащую символ кавычки, а это может произойти значительно позже. К счастью, в той же строчке обычно сразу возникает синтаксическая ошибка, а Perl достаточно сообразителен и сообщает место, где, по его мнению, может быть начало незавершенной строки.

Кроме перечисленных выше escape-последовательностей, начинающихся с обратным слэшем, строки, заключенные в двойные кавычки, подвергаются *интерполяции переменных* скалярными и списочными значениями. Это означает, что значения некоторых переменных можно вставлять непосредственно в строковый литерал. По сути это просто удобная форма конкатенации строк.[1] Интерполяция переменных может выполняться для скалярных переменных, целых массивов (но не хешей), отдельных элементов массива или хеша либо срезов (множественных индексов) массива или хеша. Интерполяция чего-либо другого не производится. Иными словами, интерполировать можно только выражения, начинающиеся с `$` или `@`, поскольку это те два символа (наряду с обратным слэшем), которые ищет анализатор строк. Внутри строк литерал `@`, не являющийся частью идентификатора массива или среза, но за которым следует буквенно-цифровой символ, должен быть предварен обратным слэшем (`\@`), иначе возникает ошибка компиляции. Хотя хеш в целом, обозначаемый с помощью `%`, не может интерполироваться в строку, отдельные значения хеша или его срезы допускаются, поскольку начинаются с `$` и `@`, соответственно.

[1] Если включен вывод предупредительных сообщений, то Perl может сообщать об интерполяции в строки неопределенных значений как об операциях конкатенации и объединения, хотя вы не используете в этих местах такие операторы. Однако их создал там вместо вас компилятор.

Следующий фрагмент кода выводит "Цена $100.":

```
$Price = '$100';                    # не интерполируется
print "Цена $Price.\n";             # интерполируется
```

Как и в некоторых оболочках, можно заключить идентификатор в фигурные скобки, чтобы отличить его от последующих буквенно-цифровых символов: "How ${verb}able!". Идентификатор в таких скобках преобразуется в строку, как всякий отдельный идентификатор в индексе хеша. Например:

```
$days{'Feb'}
```

можно записать как:

```
$days{Feb}
```

и наличие кавычек будет предполагаться. Более сложный текст в индексе интерпретируется как выражение и должен быть заключен в кавычки:

```
$days{'February 29th'}   # Ok.
$days{"February 29th"}   # Тоже ok. "" не надо интерполировать.
$days{ February 29th }   # НЕВЕРНО, влечет ошибку анализатора.
```

В частности, всегда нужно использовать кавычки в таких срезах, как:

```
@days{'Jan','Feb'}       # Ok.
@days{"Jan","Feb"}       # Тоже ok.
@days{ Jan,  Feb }       # Не совсем (ошибка при use strict)
```

Помимо индексов интерполируемых переменных массивов и хешей, нет нескольких уровней интерполяции. В противоположность ожиданиям программирующих для оболочек, обратные кавычки не приводят к интерполяции внутри двойных кавычек, и одинарные кавычки не влекут вычисление переменных, когда используются внутри двойных кавычек. Интерполяция является очень мощным средством, но в Perl находится под жестким контролем. Она происходит только в двойных кавычках и некоторых других операциях типа двойных кавычек, о которых мы расскажем в следующем разделе:

```
print "\n";              # Ok, вывод перевода строки.
print  \n ;              # НЕВЕРНО, нет интерполируемого контекста.
```

Выберите свои кавычки

Хотя обычно мы рассматриваем кавычки как литеральные значения, в Perl они действуют скорее как операторы, предоставляя различные возможности интерполяции и поиска по шаблону. Для этих режимов Perl предоставляет возможность выбора собственных символов вместо кавычек, а также более общий способ выбора символа кавычек для каждого из этих режимов. В таблице 2.3 показано, что вместо / можно указать любой буквенно-цифровой символ, отличный от пробельного. (Символы перевода строки и пробела

больше нельзя использовать в качестве разделителей, хотя в старых версиях Perl это разрешалось.)

Таблица 2.3. Структуры «закавычивания»

Пользовательские	Общие	Значение	Интерполируется
`''`	`q//`	Строка-литерал	Нет
`""`	`qq//`	Строка-литерал	Да
`` `` ``	`qx//`	Выполнение команды	Да
`()`	`qw//`	Список слов	Нет
`//`	`m//`	Поиск по шаблону	Да
`s///`	`s///`	Замена по шаблону	Да
`y///`	`tr///`	Транслирование символов	Нет
`""`	`qr//`	Регулярное выражение	Да

Некоторые из обозначений являются просто синтаксическими послаблениями, позволяющими избежать ввода многочисленных символов обратного слэша в строки, заключенные в кавычки, особенно при поиске по шаблону, когда можно запутаться в обычных символах слэша и обратного слэша.

Если выбрать в качестве ограничителя одинарные кавычки, интерполяции переменных не происходит даже в тех форматах, которые обычно интерполируются. Если открывающим ограничителем является открывающая круглая, квадратная, фигурная или угловая скобка, то закрывающим ограничителем будет соответствующий закрывающий символ. (Встраиваемые ограничители должны иметь попарное соответствие.) Примеры:

```
$single = q!Я сказал: "Ты сказал: `Она сказала это.`"!;

$double = qq(Можем мы получить какую-нибудь "хорошую" $variable?);
$chunk_of_code = q {
    if ($condition) {
        print "Гоча!";
    }
};
```

Последний пример показывает, что можно использовать пробельный символ между спецификатором кавычек и их открывающим символом. В двухэлементных конструкциях, таких как `s///` и `tr///`, когда первая пара кавычек служит парой скобок, вторая часть получает свой собственный символ начальной кавычки. В действительности вторая пара не обязательно должна совпадать с первой, поэтому можно писать такие вещи, как `s<foo>(bar)` или `tr(a-f)[A-F]`. Поскольку между двумя внутренними символами кавычек допускается наличие пробельного символа, последний пример можно записать даже так:

```
tr (a-f)
   [A-F];
```

Однако пробельные символы недопустимы, когда в качестве кавычек выступает #. q#foo# разбирается анализатором как строка 'foo', а q #foo# — как оператор кавычек q, за которым следует комментарий. Символ-ограничитель этого оператора будет взят со следующей строки. Комментарии можно помещать в середину двухэлементных конструкций, что позволяет писать:

```
s {foo}   # Заменить foo
  {bar};  #      на bar.

tr [a-f]  # Перевести шестнадцатеричные цифры из нижнего регистра
   [A-F]; #                                          в верхний
```

Или опустите кавычки полностью

Имя, для которого не находится иной грамматической интерпретации, обрабатывается так, как если бы являлось строкой, заключенной в кавычки. Такие имена называют *голыми словами* (*barewords*).[1] Так же как дескрипторы файлов и метки, голые слова[2], состоящие исключительно из букв нижнего регистра, могут вступать в конфликт со словами, которые будут зарезервированы в будущем. Если включен вывод предупредительных сообщений, Perl выводит предупреждения по поводу голых слов. Например,

```
@days = (Mon,Tue,Wed,Thu,Fri);
print STDOUT hello, ' ', world, "\n";
```

записывает в массив @days сокращенные названия дней недели и выводит hello world и символ перевода строки на STDOUT. Если опустить дескриптор файла, то Perl попытается интерпретировать в качестве дескриптора файла hello, что приведет к синтаксической ошибке. Поскольку применение голых слов чревато возникновением ошибок, некоторые программисты склонны их избегать вовсе. Перечисленные выше операторы кавычек предоставляют много удобных форм, в том числе конструкцию qw// «заключения слов в кавычки», которая точно заключает в кавычки список слов, разделенных пробелами:

```
@days = qw(Mon Tue Wed Thu Fri);
print STDOUT "hello world\n";
```

Можно пойти и на то, чтобы вообще запретить использование голых слов. Если сказать:

```
use strict 'subs';
```

---

[1] Имена переменных, дескрипторы файлов, метки и тому подобное не считаются голыми словами, поскольку их значение определяется предшествующим или последующим маркерами (или ими обоими). Имена с предварительными объявлениями, такие как подпрограммы, тоже не являются голыми словами. Голым слово является только тогда, когда у анализатора нет к нему никакого ключа.

[2] Называемые также *простыми*. — *Примеч. перев.*

то каждое голое слово приводит к возникновению ошибки времени компиляции. Это ограничение действует на протяжении всей охватывающей области видимости. Во внутренней области видимости можно отменить действие этой директивы, сказав:

```
no strict 'subs';
```

Обратите внимание, что голые идентификаторы в конструкциях типа:

```
"${verb}able"
$days{Feb}
```

не считаются голыми словами, поскольку их предоставляет явное правило, а не «отсутствие иной грамматической интерпретации».

Имя без кавычек, за которым следуют два двоеточия, например `main::` или `Dog::`, всегда считается именем пакета. На этапе компиляции Perl преобразует могущее оказаться голым словом `Camel::` в строку `Camel`, поэтому при наличии директивы `use strict` его применение не вызовет осуждения.

Интерполяция величин, являющихся массивами

Переменные, являющиеся массивами, интерполируются в строки, заключенные в двойные кавычки, путем объединения всех элементов массива через разделитель, заданный в переменной `$"`[1] (по умолчанию содержащей пробел). Эквивалентны следующие формы:

```
$temp = join( $", @ARGV );
print $temp;

print "@ARGV";
```

В шаблонах поиска, тоже подвергающихся интерполяции двойных кавычек, возникает досадная двусмысленность: следует ли `/$foo[bar]/` интерпретировать как `/${foo}[bar]/` (где `[bar]` является классом символов в регулярном выражении) или как `/${foo[bar]}/` (где `[bar]` является индексом массива `@foo`)? Если `@foo` не существует, то это, очевидно, класс символов. Если `@foo` существует, Perl пытается угадать, чем является `[bar]`, и почти всегда с успехом.[2] Если он угадывает неверно или вы страдаете паранойей, то можно принудительно задать правильную интерпретацию с помощью фигурных скобок, как показано выше. Даже если вы всего лишь проявляете благоразумие, это может быть неплохой мыслью.

---

[1] `$LIST_SEPARATOR`, если используется модуль English, поставляемый с Perl.

[2] Полностью описывать алгоритм угадывания утомительно, но в основе лежит сравнение взвешенного среднего всего, что похоже на классы символов (`a-z`, `\w`, начальный `^`), с тем, что похоже на выражения (переменных и зарезервированных слов).

Встроенные документы («here» documents)

Строчно-ориентированная форма расстановки кавычек основана на синтаксисе *встроенных документов* (*here-documents*) оболочки Unix. Под строчной ориентированностью имеется в виду, что ограничителями являются не символы, а строки. Начальным ограничителем является текущая строка, а конечным – строка, которую вы задаете. Вслед за << задается строка, которая должна завершить заключаемый в кавычки текст, и все строки, следующие за текущей, но исключая завершающую, становятся частью строки. Завершающая строка может быть идентификатором (словом) или некоторым текстом, заключенным в кавычки. В случае применения кавычек их тип определяет обработку текста, как и при обычном использовании кавычек. Идентификатор без кавычек действует так, как если бы был заключен в двойные кавычки. Идентификатор, которому предшествует обратная косая черта, действует так, как если бы он был заключен в одинарные кавычки (для совместимости с синтаксисом оболочки). Между << и идентификатором без кавычек не должно быть пробела, хотя пробельный символ разрешен, если указывается не голый идентификатор, а строка в кавычках. (Если вставить пробел, то он будет рассматриваться как нулевой идентификатор, что допустимо, но осуждается, и соответствует первой пустой строке – см. первое Ура! в нижеследующем примере.) Завершающая строка должна задаваться в чистом виде, без кавычек и лишних пробельных символов по концам в последней строчке.

```
print <<EOF;    # то же, что в прежнем примере
Цена $Price.
EOF

print <<"EOF";  # то же, что выше, но с явными кавычками
Цена $Price.
EOF

print <<'EOF';    # Одинарные кавычки
All things (e.g. a camel's journey through
A needle's eye) are possible, it's true.
But picture how the camel feels, squeezed out
In one long bloody thread, from tail to snout.
                                -- C.S. Lewis
Да, это правда, - многое, и даже все - возможно
(Верблюд преодолеет узкие врата игольного отверстия).
Но каково быть превращенным в кровавое и длинное страдание
От морды до хвоста?

EOF

print << x 10;    # вывести следующую строку 10 раз
Верблюды идут!  Ура!  Ура!
```

```
print <<"" x 10;  # более предпочтительная запись того же
Верблюды идут!  Ура!  Ура!

print <<`EOC`;    # выполнить команды
echo эй, там!
echo смотри туда!
EOC

print <<"верблюд одногорбый", <<"camelid";   # можно собрать в кучу
Я сказал верблюд двугорбый.
верблюд одногорбый
Она сказала лама.
camelid

funkshun(<<"ЭТО", 23, <<'ТО');   # то, что они в скобках, не имеет значения
Вот строка
или две строки.
ЭТО
Вот еще одна.
ТО
```

Не забывайте ставить точку с запятой в конце команды, поскольку Perl не знает, что вы не собираетесь делать следующее:

```
print <<'odd'
2345
odd
    + 10000;   # выводит 12345
```

Для того чтобы встроенный документ имел такой же отступ, как в остальном коде, надо вручную удалить пробельные символы из каждой строки:

```
($quote = <<'QUOTE') =~ s/^\s+//gm;
    The Road goes ever on and on,
    down from the door where it began.[1]
QUOTE
```

Можно даже заполнить массив строчками встроенного документа, например:

```
@sauces = <<End_Lines =~ m/(\S.*\S)/g;
   обычный томатный
   томатный со специями
   зеленый чили
   сырный "зеленый"
   с белым вином
End_Lines
```

[1] Из «Властелина колец» Дж. Р. Р. Толкиена. «Дорога вдаль и вдаль ведет, в ее начале – мой порог» (пер. Г. Виноградов). – *Примеч. науч. ред.*

Литералы v-строк

Литерал, начинающийся с v, за которым следуют одно или более целых чисел, разделяемых точками, рассматривается как строковый литерал, состоящий из символов с указанными порядковыми номерами:

```
$crlf = v13.10;          # ASCII-символы возврата каретки, перевода строки
```

Такие литералы называются *v-строками*, сокращенно от «векторных строк» или «строк версии» или чего-нибудь еще, что можно вообразить себе начинающимся с «v» и содержащим список целых чисел. Они предоставляют альтернативный и более доходчивый способ создания строк, когда требуется задать численные значения для каждого символа. Так, например, v1.20.300.4000 служит более приятным средством создания строкового значения, чем любое из следующих:

```
"\x{1}\x{14}\x{12c}\x{fa0}"
pack("U*", 1, 20, 300, 4000)
chr(1) . chr(20) . chr(300) . chr(4000)
```

Если в таком литерале две или более точки (три или более целых числа), то ведущее v можно опустить.

```
print v9786;             # выводит SMILEY в кодировке UTF-8, "\x{263a}"
print v102.111.111;      # выводит "foo"
print 102.111.111;       # то же самое

use 5.6.0;               # требует указанной версии Perl (или более новой)

$ipaddr = 204.148.40.9;  # IPv4-адрес oreilly.com
```

При помощи v-строк удобно представлять IP-адреса и номера версий. В частности, поскольку в наше время символы могут иметь порядковые значения, большие 255, v-строки дают способ представления номеров версий любого размера, которые можно корректно сравнивать с помощью обычного сравнения строк.

Номера версий и IP-адреса, хранящиеся в v-строках, не предназначены для чтения человеком, поскольку отдельные целые числа хранятся в виде произвольных символов. Чтобы создать нечто читаемое, используйте флаг v в маске printf, например "%vd", согласно описанию sprintf в главе 29 «Функции». Дополнительные сведения о строках Unicode можно найти в главе 15 «Unicode», а о директиве use bytes — в главе 31 «Модули прагм»; о сравнении строк версий с применением операторов сравнения строк и представлении адресов IPv4 см. описание gethostbyaddr в главе 29 «Функции».

Другие литеральные лексемы

Все идентификаторы, начинающиеся и оканчивающиеся удвоенным символом подчеркивания, зарезервированы в Perl для особых синтаксических целей. Двумя такими особыми литералами являются __LINE__ и __FILE__, кото-

рые представляют текущие номер строки и имя файла в данной точке вашей программы. Их можно использовать только как отдельные лексемы, они не интерполируются в строки. Аналогично, __PACKAGE__ представляет собой имя пакета, в который компилируется текущий код. Если текущего пакета нет (согласно пустой директиве package;), __PACKAGE__ имеет неопределенное значение. Лексему __END__ (или же символы Control-D или Control-Z) можно использовать для обозначения логического окончания сценария до реального конца файла. Любой следующий за ними текст игнорируется, но может быть прочитан с помощью дескриптора файла DATA.

Лексема __DATA__ действует аналогично лексеме __END__, но открывает дескриптор файла DATA в пространстве имен текущего пакета, поэтому в файлах, загружаемых посредством require, могут одновременно быть открыты собственные дескрипторы файлов DATA. За дополнительными сведениями обратитесь к описанию DATA в главе 28 «Специальные имена».

Контекст

Мы уже видели несколько термов, которые могут порождать списочный контекст. Однако прежде чем продолжить обсуждение термов, следует договориться по поводу самого понятия *контекста*.

Скалярный и списочный контексты

Всякая операция[1], вызываемая в сценарии Perl, выполняется в конкретном контексте, и характер ее действия может зависеть от требований этого контекста. Есть два основных контекста: скалярный и списочный. Например, присвоение скалярной переменной или скалярному элементу массива или хеша приводит к вычислению правой части в *скалярном контексте*:

```
$x         = funkshun();  # скалярный контекст
$x[1]      = funkshun();  # скалярный контекст
$x{"ray"}  = funkshun();  # скалярный контекст
```

Однако присвоение массиву или хешу либо срезу того или другого вычисляет правую часть в *списочном контексте*, даже если в срезе выбран только один элемент:

```
@x         = funkshun();  # списочный контекст
@x[1]      = funkshun();  # списочный контекст
@x{"ray"}  = funkshun();  # списочный контекст
%x         = funkshun();  # списочный контекст
```

[1] В данном случае мы свободно пользуемся термином «операция» для обозначения либо оператора, либо терма. Различие между двумя понятиями размывается, если говорить о функциях, анализируемых как термы, но выглядящих, как унарные операторы.

Присвоение списку скаляров тоже обеспечивает списочный контекст в правой части, даже если в списке только один элемент:

```
($x,$y,$z) = funkshun();  # списочный контекст
($x)       = funkshun();  # списочный контекст
```

Правила остаются неизменными при объявлении переменной путем модификации терма с помощью `my` или `our`, поэтому:

```
my $x      = funkshun();  # скалярный контекст
my @x      = funkshun();  # списочный контекст
my %x      = funkshun();  # списочный контекст
my ($x)    = funkshun();  # списочный контекст
```

Плохо вам будет, пока вы не усвоите различие между скалярным и списочным контекстами, поскольку некоторые операторы (такие, как гипотетическая функция `funkshun()`, приведенная нами выше) знают, в каком контексте они находятся, и возвращают список в контекстах, требующих списка, и скалярное значение в контекстах, требующих скаляра. (Если это относится к операции, то об этом будет сказано в документации по этой операции.) На компьютерном жаргоне говорят, что операции *перегружаются* в зависимости от типа возвращаемого значения. Однако это очень простой вид перегрузки, основывающийся только на различии между единственным и множественными значениями и ни на чем ином.

Если некоторые операторы реагируют на контекст, то, очевидно, должно быть что-то, окружающее их, предоставляющее этот контекст. Мы показали, что присвоение может предоставлять контекст своему правому операнду, но это не должно удивлять, поскольку все операторы предоставляют тот или иной контекст для каждого из своих операндов. Что действительно необходимо знать, так это то, *какие* операторы *какой* контекст предоставляют своим операндам. Оказывается, отличить операторы, предоставляющие списочный контекст, не трудно, поскольку в их синтаксических описаниях указан `LIST`. Все остальное предоставляет скалярный контекст. Обычно это вполне интуитивно.[1] При необходимости можно установить скалярный контекст для аргумента в середине `LIST` с помощью псевдофункции `scalar`. В Perl нет средства установки списочного контекста в скалярном контексте, поскольку там, где требуется списочный контекст, он уже обеспечен с помощью `LIST` какой-либо управляющей функции.

Скалярный контекст может быть подразделен на строковый, числовой и произвольный контексты. В противоположность различению скалярного и спискового контекстов операции не имеют возможности и необходимости знать, в котором из скалярных контекстов они находятся. Они просто воз-

---

[1] Обратите, однако, внимание на то, что списочный контекст *`LIST`* может распространяться при вызове подпрограмм, поэтому не всегда очевидно, будет ли данная команда выполняться в скалярном или списочном контексте. Программа может определить свой контекст в подпрограмме с помощью функции `wantarray`.

вращают требуемое скалярное значение и предоставляют Perl переводить числа в строки в строковом контексте и строки в числа в числовом контексте. В некоторых скалярных контекстах безразлично, возвращается ли строка, число или ссылка, поэтому преобразование не осуществляется. Это происходит, например, в случае присвоения значения другой переменной. Новая переменная просто получает тот же подтип, что и старое значение.

Логический (булев) контекст

Другим особым произвольным скалярным контекстом является *булев контекст*. Булев контекст представляет собой просто любое место, где вычисляется значение выражения и проверяется его истинность. Когда в этой книге мы говорим «true» и «false», то имеем в виду используемое в Perl техническое определение: скалярное значение является истинным, если оно не равно нулевой строке `""` или числу 0 (или его строковому эквиваленту, `"0"`). Ссылка всегда истинна, поскольку представляет адрес, не могущий быть 0. Неопределенное значение (часто обозначаемое `undef`) всегда ложно, поскольку является `""` или 0, в зависимости от того, рассматривается ли это значение как строка или как число. (Списочные величины не имеют логического значения, поскольку никогда не создаются в скалярном контексте!)

Поскольку булев (логический) контекст является безразличным контекстом, он не вызывает преобразования скаляров, хотя, конечно, сам скалярный контекст налагается на любой операнд, различающий контекст. А для многих различающих операндов скаляр, создаваемый ими в скалярном контексте, представляет собой некоторое булево значение. Это значит, что операторы, создающие список в списочном контексте, могут быть использованы для проверки на true/false в булевом контексте. Например, в списочном контексте, производимом оператором `unlink`, имя массива порождает список его значений:

```
unlink @files;      # Удалить все файлы, игнорируя ошибки.
```

Но если использовать массив в условном операторе (то есть в булевом контексте), то массив определяет, что находится в скалярном контексте, и возвращает число элементов в массиве, которое является истинным значением, пока в массиве есть какие-нибудь элементы. Поэтому, если желательно вывести предупреждения для всех файлов, которые не могли быть удалены, можно написать такой цикл:

```
while (@files) {
    my $file = shift @files;
    unlink $file or warn "Невозможно удалить $file: $!\n";
}
```

Здесь `@files` вычисляется в логическом контексте, предоставляемом командой `while`, поэтому Perl вычисляет сам массив и проверяет, является ли он истинным массивом или ложным массивом. Истинным массивом он остается, пока в нем содержатся имена файлов, но становится ложным, как только из него выталкивается последнее имя файла. Обратите внимание, что

остается справедливым сказанное нами выше. Несмотря на то что массив содержит (и может порождать) списочное значение, мы не вычисляем списочное значение в скалярном контексте. Мы сообщаем массиву, что он скаляр, и спрашиваем, что он сам о себе думает.

Не соблазнитесь использованием для этого `defined @files`. Ничего не получится, потому что функция `defined` определяет, является ли скаляр равным `undef`, но массив – это не скаляр. Достаточно простой логической проверки.

Пустой контекст

Еще одним специфическим видом скалярного контекста является *пустой контекст*. В этом контексте не только безразлично, какого типа значение возвращается, но и *не требуется* возвращать значение. С точки зрения работы функций, он не отличается от обычного скалярного контекста. Но если включен вывод предупредительных сообщений, то компилятор Perl предупреждает о применении выражения без побочных эффектов в таком месте, где не требуется значения, например в команде, не возвращающей значения. Например, если в качестве команды используется строка:

```
"Camel Lot";
```

можно получить предупреждение такого вида:

```
Useless use of a constant in void context in myprog line 123;
[бесполезное использование константы в пустом контексте в myprog, строка 123]
```

Интерполирующий контекст

Выше мы говорили, что в литеральных строках в двойных кавычках производится интерполяция обратного слэша и переменных, но интерполирующий контекст (часто называемый контекстом двойных кавычек, поскольку не все могут выговорить «интерполирующий») применяется не только к строкам в двойных кавычках. Среди других конструкций двойных кавычек находятся обобщенный оператор обратных кавычек `qx//`, оператор поиска по шаблону `m//`, оператор подстановки `s///` и оператор заключения в кавычки регулярного выражения `qr//`. Оператор подстановки производит интерполяцию в левой части перед поиском по шаблону, а затем интерполяцию в правой части при каждом нахождении соответствия левой части.

Интерполирующий контекст возникает только внутри кавычек или конструкций, работающих как кавычки, поэтому, может быть, не совсем справедливо называть его контекстом в таком же смысле, как скалярный и списочный контексты. (А может быть, и справедливо.)

Списочные значения и массивы

Теперь, когда мы обсудили контексты, можно поговорить о списочных литералах и их поведении в контексте. Вы уже видели несколько списочных ли-

тералов. Списочные литералы обозначаются путем разделения отдельных значений запятыми (и заключения списка в круглые скобки, если этого требует приоритетность). Поскольку (почти) никогда не вредно добавить пару лишних скобок, синтаксическая диаграмма списочного значения обычно обозначается так:

```
(LIST)
```

Равнее мы говорили, что LIST в синтаксическом описании указывает на нечто, предоставляющее списочный контекст для своих аргументов, но голый списочный литерал является некоторым исключением из этого правила, поскольку предоставляет списочный контекст своим аргументам, только когда список в целом находится в списочном контексте. Значениями списочного литерала в списочном контексте являются просто значения аргументов в заданном порядке. В качестве необычной разновидности терма в выражении списочный литерал просто проталкивает ряд временных значений на стек Perl, откуда позже они изымаются оператором, требующим список.

Однако в скалярном контексте списочный литерал ведет себя не так, как LIST, поскольку не предоставляет списочный контекст своим значениям. Вместо этого он просто вычисляет каждый из своих аргументов в скалярном контексте и возвращает значение последнего элемента. Это происходит потому, что он представляет собой в действительности просто замаскированный оператор запятой из C, который является бинарным оператором, всегда отбрасывающим значение слева и возвращающим значение справа. Говоря на языке обсуждавшегося нами выше, левая сторона оператора запятой, в сущности, предоставляет пустой контекст. Поскольку оператор запятой обладает левой ассоциативностью, то ряд значений, разделяемых запятыми, приводит в итоге к последнему значению, т. к. последняя запятая отбрасывает все, что произвели предыдущие запятые. Разница между контекстами видна из того, что списочное присвоение

```
@stuff = ("один", "два", "три");
```

присваивает все списочное значение массиву @stuff, но скалярное присвоение

```
$stuff = ("один", "два", "три");
```

присваивает только значение «три» переменной $stuff. Подобно упоминавшемуся выше массиву @files, оператор запятой знает, находится он в скалярном или списочном контексте, и действует соответствующим образом.

Можно повторить, что списочное значение отлично от массива. Действительная переменная массива знает также свой контекст и в списочном контексте возвращает свой внутренний список значений как списочный литерал. Но в скалярном контексте она возвращает только размер массива. В следующем примере переменной $stuff присваивается значение 3:

```
@stuff = ("один", "два", "три");
$stuff = @stuff;
```

Если вы ожидали значение "три", то, вероятно, сделали неверное обобщение, предположив, что Perl использует правило оператора запятой, чтобы отбросить все, кроме одного, временные значения, которые помещены @stuff на стек. Но дело обстоит не так. Массив @stuff не помещал все свои значения на стек. В действительности он не помещал на стек ни одного своего значения. Он поместил лишь одну величину, размер массива, поскольку *знал*, что находится в скалярном контексте. Ни один терм или оператор в скалярном контексте не помещает на стек список. Вместо этого он помещает в стек один скаляр (каким бы непривычным это ни показалось), который едва ли окажется последним значением списка, которое он *возвратил бы* в списочном контексте, потому что последнее значение вряд ли окажется самым полезным значением в скалярном контексте. Уловили? (Если нет, лучше перечитайте этот абзац еще раз, поскольку это важно.)

Теперь вернемся к настоящим LIST — тем, которые образуют списочный контекст. До сих пор мы представляли дело так, будто списочные литералы являются просто списками литералов. Но так же как строковый литерал может интерполировать другие подстроки, списочный литерал может интерполировать другие подсписки. В списке может находиться любое выражение, возвращающее значения. Используемые при этом значения могут быть скалярными или списочными, но все они становятся частью нового списочного значения, потому что LIST производит автоматическую интерполяцию подсписков. Это означает, что при вычислении LIST каждый элемент списка вычисляется в списочном контексте, и результирующее списочное значение интерполируется в LIST, как если бы каждый отдельный элемент был членом LIST. Поэтому массивы утрачивают свою индивидуальность в LIST.[1] Список

```
(@stuff,@nonsense,funkshun())
```

содержит элементы @stuff, за которыми следуют элементы @nonsense, а за ними следуют те значения, которые подпрограмма &funkshun сочтет нужными возвратить в списочном контексте. Обратите внимание, что все они могут интерполироваться нулевым (пустым) списком, и в этом случае все происходит так, как если бы в этом месте не интерполировался никакой массив или вызов функции. Собственно нулевой список представляется литералом (). Как и для нулевого массива, который интерполируется как нулевой список и потому фактически игнорируется, интерполяция нулевого списка в другой список не оказывает никакого эффекта. Поэтому (((),(),())) эквивалентно ().

Следствием этого правила является то, что можно поместить необязательную запятую в конце любого списочного значения. Это упростит в будущем помещение дополнительных элементов вслед за последним:

```
@releases = (
    "альфа",
```

[1] Некоторым кажется, что данное обстоятельство представляет собой проблему, но это не так. Всегда можно интерполировать ссылку на массив, если вы не хотите, чтобы он утратил свою индивидуальность. См. главу 8 «Ссылки».

```
    "бета",
    "гамма",
);
```

Можно вообще отказаться от запятых: другим способом задания списка литералов является упомянутый выше синтаксис qw (заключение в кавычки слов). Эта конструкция эквивалентна расщеплению строки в одинарных кавычках по пробельным символам. Например:

```
@froots = qw(
    яблоко      банан       карамболь
    кокос       гуава       кумкват
    мандарин    нектарин    персик
    груша       хурма       слива
);
```

(Обратите внимание, что эти круглые скобки ведут себя не как обычно, а как кавычки. С таким же успехом это могли быть угловые или фигурные скобки или косая черта. Но круглые скобки красивее.)

Со списочным значением можно использовать индексы, как в обычном массиве. Для избежания неоднозначности нужно заключать список в круглые скобки (настоящие). Хотя таким способом часто из списка извлекаются отдельные значения, на самом деле это срез списка, поэтому синтаксис такой:

```
(LIST)[LIST]
```

Примеры:

```
# stat возвращает списочное значение.
$modification_time = (stat($file))[9];

# ЗДЕСЬ СИНТАКСИЧЕСКАЯ ОШИБКА.
$modification_time = stat($file)[9];  # ОЙ, СКОБКИ ЗАБЫЛИ

# Найти шестнадцатеричную цифру.
$hexdigit = ('a','b','c','d','e','f')[$digit-10];

# "Оператор обратной запятой".
return (pop(@foo),pop(@foo))[0];

# Получить несколько значений как срез.
($day, $month, $year) = (localtime)[3,4,5];
```

Присвоение спискам

Присвоение списку можно осуществлять, только если допустимо присвоение для каждого элемента списка:

```
($a, $b, $c) = (1, 2, 3);
($map{red}, $map{green}, $map{blue}) = (0xff0000, 0x00ff00, 0x0000ff);
```

Можно осуществлять присвоение элементам undef в списке. Это удобно, если нужно отбросить некоторые значения, возвращаемые функцией:

```
($dev, $ino, undef, undef, $uid, $gid) = stat($file);
```

Последним элементом списка может быть массив или хеш:

```
($a, $b, @rest) = split;
my ($a, $b, %rest) = @arg_list;
```

На самом деле можно поместить массив или хеш в любом месте списка, которому производится присвоение, но первый массив или хеш, встретившийся в списке, поглотит все оставшиеся значения, и все последующие элементы списка получат неопределенные значения. Это можно использовать в local или my, когда нужно, чтобы инициализированные массивы остались пустыми.

Можно даже осуществлять присвоение пустому списку:

```
() = funkshun();
```

Это приводит к вызову функции в списочном контексте с отбрасыванием возвращаемых значений. Если бы функция была вызвана без присвоения, то вызов произошел бы в пустом контексте, который является разновидностью скалярного контекста, при этом функция могла бы повести себя совершенно иначе.

Присвоение списку в скалярном контексте возвращает число элементов, образованных выражением в *правой* части присвоения:

```
$x = ( ($a, $b) = (7,7,7) );   # присваивает $x 3, а не 2
$x = ( ($a, $b) = funk() );    # присваивает $x количество значений, возвращаемых
funk()
$x = ( () = funk() );          # тоже присваивает $x число значений, возвращаемых
funk()
```

Это удобно в случае присвоения списку в булевом контексте, поскольку большинство списочных функций возвращают по завершении нулевой список, который при присвоении порождает 0, интерпретируемый как false. Вот как можно использовать это в цикле while:

```
while (($login, $password) = getpwent) {
    if (crypt($login, $password) eq $password) {
        print "$login имеет ненадежный пароль!\n";
    }
}
```

Размер массива

Число элементов в массиве @days можно определить, вычисляя @days в скалярном контексте, например:

```
@days + 0;       # неявная установка для @days скалярного контекста
scalar(@days)    # явная установка для @days скалярного контекста
```

Обратите внимание, что это действует только для массивов. В целом для списочных значений это не действует. Как уже говорилось, список с разделителем-запятой, вычисляемый в скалярном контексте, возвращает последнее значение, как оператор запятой в C. Но поскольку в Perl практически никогда не требуется знать размер списка, из-за этого не возникает проблем.

Со скалярным вычислением `@days` тесно связана переменная `$#days`. Она возвращает индекс последнего элемента массива, на единицу меньший размера массива, поскольку (обычно) имеется 0-й элемент. Присвоение переменной `$#days` изменяет размер массива. Укорачивание массива с помощью этого метода уничтожает лишние значения. Можно добиться некоторого увеличения эффективности программы, заранее увеличивая размер массива, который должен разрастись. (Расширить массив можно и присвоением значения элементу, находящемуся за его пределами.) Можно произвести полное усечение массива, присвоив ему нулевой список `()`. Следующие две команды эквивалентны:

```
@whatever = ();
$#whatever = -1;
```

Всегда выполняется и следующее равенство:

```
scalar(@whatever) == $#whatever + 1;
```

Усечение массива не высвобождает занимаемую им память. Чтобы вернуть память в пул памяти процесса, нужно использовать `undef(@whatever)`. Вернуть память в системный пул, вероятно, не удастся, поскольку немногие операционные системы поддерживают это.

Хеши

Как было сказано ранее, хеш является просто необычным видом массива, поиск значений в котором осуществляется с помощью ключевых строк, а не чисел. Хеш ассоциирует ключи со значениями, поэтому хеши часто называют *ассоциативными массивами* (те, кому не лень вводить столько букв с клавиатуры).

На практике в Perl нет такой вещи, как хеш-литерал, но если присвоить хешу обычный список, то каждая пара значений в списке образует одну связь ключ/значение:

```
%map = ('red',0xff0000,'green',0x00ff00,'blue',0x0000ff);
```

Такой же результат получается после команд:

```
%map = ();              # сначала очистить хеш
$map{red}   = 0xff0000;
$map{green} = 0x00ff00;
$map{blue}  = 0x0000ff;
```

Часто код легче читать, если использовать оператор => в парах ключ/значение. Оператор => служит просто синонимом запятой, но зрительно более отчетлив и заключает в кавычки голые идентификаторы в своей левой части (как идентификаторы в фигурных скобках выше), что делает его удобным для нескольких видов операций, включая инициализацию переменных типа хеш:

```
%map = (
    red   => 0xff0000,
    green => 0x00ff00,
    blue  => 0x0000ff,
);
```

или инициализацию ссылок на анонимные хеши, используемые как записи:

```
$rec = {
    NAME  => 'John Smith',
    RANK  => 'Captain',
    SERNO => '951413',
};
```

или применение именованных параметров для вызова сложных функций:

```
$field = radio_group(
             NAME      => 'животные',
             VALUES    => ['верблюд', 'лама', 'баран', 'волк'],
             DEFAULT   => 'верблюд',
             LINEBREAK => 'true',
             LABELS    => \%animal_names,
         );
```

Но мы снова забегаем вперед. Вернемся к хешам.

Переменную типа хеш (%hash) можно использовать в списочном контексте, и в этом случае она интерполирует все свои пары ключ/значение в список. То, что хеш инициализировался в определенном порядке, не означает, что его значения будут в том же порядке возвращаться. Внутренне хеши реализуются с применением хеш-таблиц для быстрого поиска, в результате чего порядок, в котором запоминаются элементы, зависит от внутренней хеш-функции, используемой для расчета позиции в хеш-таблице, а не чего-либо другого. Поэтому записи возвращаются в порядке, кажущемся случайным. (Оба элемента каждой пары ключ/значение возвращаются, конечно, в правильном порядке.) Примеры способов упорядочения вывода можно найти в описании функции keys в главе 29 «Функции».

При вычислении переменной, представляющей собой хеш, в скалярном контексте, значение true возвращается, если в хеше есть хотя бы одна пара ключ/значение. Если пары ключ/значение есть, возвращаемое значение является строкой, состоящей из числа использованных блоков и числа выделенных блоков, разделенных косой чертой. В основном это полезно только для того, чтобы обнаружить, что алгоритм хеширования Perl (встроенный

в него) плохо работает на ваших данных. Например, если вы поместите в хеш 10 000 значений, но при вычислении %HASH в скалярном контексте получите 1/8, это будет означать, что только один из восьми блоков задействован. Вероятно, в этом одном блоке содержатся все 10 000 ваших элементов. Скорее всего, этого не произойдет.

Чтобы определить количество ключей в хеше, вызывайте функцию keys в скалярном контексте: scalar(keys(%HASH)).

Многомерный хеш можно эмулировать путем задания в фигурных скобках нескольких ключей, разделенных запятыми. Перечисленные ключи конкатенируются вместе, при этом в качестве разделителя выступает содержимое $; ($SUBSCRIPT_SEPARATOR), по умолчанию равное chr(28). Получившаяся строка используется в качестве фактического ключа хеша. Две следующие строки выполняют одно и то же:

```
$people{ $state, $county } = $census_results;
$people{ join $; => $state, $county } = $census_results;
```

Эта функция первоначально была реализована для поддержки *a2p*, транслятора из *awk* в Perl. В нынешние времена вы можете просто использовать реальный (скажем, более реальный) многомерный массив, как это описывается в главе 9 «Структуры данных». Но в одном случае старый стиль все еще удобен: для хешей, связанных с файлами DBM (см. описание DB_File в главе 32 «Стандартные модули»), которые не поддерживают многомерные ключи.

Не путайте эмуляции многомерных хешей со срезами. Первое представляет скалярное значение, а второе – списочное:

```
$hash{ $x, $y, $z }      # одиночное значение
@hash{ $x, $y, $z }      # срез из трех значений
```

Таблицы имен и дескрипторы файлов

В Perl есть особый тип под названием *typeglob*, предназначенный для хранения целых записей таблицы имен. (Запись таблицы имен *foo содержит значения $foo, @foo, %foo, &foo и нескольких интерпретаций старого доброго foo.) Префиксом типа typeglob является *, поскольку он представляет все типы.

Одним из применений typeglob (или ссылок на него) является передача и запоминание дескрипторов файлов. Для того чтобы запомнить дескриптор файла, поступите так:

```
$fh = *STDOUT;
```

или используйте реальную ссылку:

```
$fh = \*STDOUT;
```

Это дает также способ создания локального дескриптора файла, например:

```
sub newopen {
    my $path = shift;
    local *FH;          # ни my(), ни our()
    open(FH, $path) or return undef;
    return *FH;         # не \*FH!
}
$fh = newopen('/etc/passwd');
```

Другие способы создания новых дескрипторов файлов приведены в описании функции `open`.

В настоящее время основным применением typeglob является создание псевдонима для одной записи таблицы имен с помощью другой записи таблицы имен. Псевдоним можно рассматривать как прозвище. Если сказать:

```
*foo = *bar;
```

то все с именем `foo` становится синонимом для соответствующего объекта, называемого `bar`. Можно создать псевдоним лишь для одной переменной из записи таблицы имен, присваивая ссылку:

```
*foo = \$bar;
```

При этом `$foo` становится псевдонимом для `$bar`, но `@foo` не становится псевдонимом для `@bar`, как и `%foo` для `%bar`. Все это оказывает воздействие только на глобальные (пакетные) переменные; доступ к лексическим переменным не может быть осуществлен через записи таблицы имен. Такое создание псевдонимов для глобальных переменных может показаться делом вовсе ненужным, однако на этой функции построен весь механизм экспорта/импорта модулей, поскольку ничто не указывает на то, что символ, для которого создан псевдоним, должен находиться в вашем пространстве имен. Команда

```
local *Here::blue = \$There::green;
```

временно делает `$Here::blue` псевдонимом для `$There::green`, но не делает `@Here::blue` псевдонимом для `@There::green` или `%Here::blue` псевдонимом для `%There::green`. К счастью, все эти сложные манипуляции с typeglob там, где нужно, скрыты от ваших глаз. Дополнительное обсуждение typeglob и проблем импорта можно найти в разделах «Ссылки на обработчики» и «Ссылки на таблицы имен» главы 8 «Ссылки», в разделе «Таблицы имен» главы 10 «Пакеты» и в главе 11 «Модули».

Операторы ввода

Есть несколько операторов ввода, которые мы опишем здесь, поскольку они анализируются как термы. Иногда их называют псевдолитералами, поскольку во многих отношениях они ведут себя, как строки в кавычках. (Операторы вывода типа `print` анализируются как списочные операторы и обсуждаются в главе 29 «Функции».)

Оператор ввода команд (обратные кавычки)

Прежде всего, имеется оператор ввода команд, известный также как оператор «обратные кавычки» (backticks), поскольку он выглядит так:

```
$info = `finger $user`;
```

Строка, заключенная в обратные кавычки (или, говоря «техническим» языком, в грависы), сначала претерпевает интерполяцию переменных, как строка в двойных кавычках. Затем результат интерпретируется системой как командная строка, а выдача команды становится значением псевдолитерала. (Это делается по подобию с аналогичным оператором оболочек Unix.) В скалярном контексте возвращается одна строка, содержащая выдачу. В списочном контексте возвращается список значений, по одному для каждой строки выдачи. (С помощью `$/` можно назначить иной символ окончания строки.)

Команда выполняется при всяком вычислении псевдолитерала. Числовой статус выполнения команды сохраняется в `$?` (см. в главе 28 «Особые имена» интерпретацию `$?`, известную также, как `$CHILD_ERROR`). В отличие от версии этой команды в *csh*, в возвращаемых данных не производится трансляция: перевод строки остается переводом строки. В отличие от всех оболочек, одинарные кавычки в Perl не скрывают от интерпретации имена переменных в команде. Чтобы передать `$` оболочке, нужно защитить этот символ обратным слэшем. Переменная `$user` в вышеприведенном примере *finger* интерполируется Perl, а не оболочкой. (Поскольку команда подвергается обработке оболочкой, возникают проблемы с защитой данных, рассматриваемые в главе 23 «Защита данных».)

Обобщенной формой «обратных кавычек» является `qx//` (от «quoted execution» – «выполнение в кавычках»), но оператор действует в точности так же, как обычные обратные кавычки. Просто вы можете выбрать свои символы в качестве кавычек. Как и в случае аналогичных псевдофункций заключения в кавычки, если в качестве разделителя выбирается одинарная кавычка, то команда не подвергается интерполяции, как в случае двойных кавычек:

```
$perl_info  = qx(ps $$);            # это $$ в Perl
$shell_info = qx'ps $$';            # это $$ в оболочке
```

Оператор ввода строки (угловых скобок)

Наиболее интенсивно из операторов ввода используется оператор ввода строки, который называют также оператором угловых скобок и функцией `readline` (поскольку внутренне он вызывает именно ее). Вычисление дескриптора файла в угловых скобках (например, `STDIN`) предоставляет следующую строку из ассоциированного дескриптора файла. (Символ перевода строки включается в состав строки, поэтому согласно критериям истинности Perl вновь введенная строка всегда является истинной, пока не будет достигнут конец файла, и тогда будет возвращено неопределенное значение, которое, к счастью, является ложным.) Обычно входное значение присваивается пере-

менной, но в одном случае происходит автоматическое присвоение. Тогда и только тогда, когда в условии цикла while нет ничего, кроме оператора ввода строки, значение автоматически присваивается специальной переменной $_. При этом проверяется, определено ли присвоенное значение. (Эта конструкция может показаться странной, но вы будете часто ее использовать, поэтому стоит ее изучить.) Во всяком случае, следующие строки эквивалентны:

```
while (defined($_ = <STDIN>)) { print $_; }   # самый длинный путь
while ($_ = <STDIN>) { print; }               # явно относительно $_
while (<STDIN>) { print; }                    # короткий способ
for (;<STDIN>;) { print; }                    # замаскированный цикл while
print $_ while defined($_ = <STDIN>);         # длинный модификатор команды
print while $_ = <STDIN>;                     # явно относительно $_
print while <STDIN>;                          # короткий модификатор команды
```

Помните, что для этого особого применения требуется цикл while. Когда оператор ввода используется в любом другом месте, результат должен присваиваться явно, если надо сохранить его значение:

```
while (<FH1> && <FH2>) { ... }       # НЕВЕРНО: отбрасывает оба ввода
if (<STDIN>)      { print; }         # НЕВЕРНО: выводит прежнее значение $_
if ($_ = <STDIN>) { print; }         # не самое оптимальное: нет проверки
определенности
if (defined($_ = <STDIN>)) { print; }# самое лучшее
```

При неявном присвоении $_ в цикле, использующем $_, это имя указывает на глобальную переменную, а не локальную для цикла while. Существующее значение $_ можно защитить так:

```
while (local $_ = <STDIN>) { print; }   # использовать локальную $_
```

Предшествующее значение восстанавливается по завершении цикла. Переменная $_ остается, однако, глобальной, поэтому функции, вызываемые изнутри цикла, могут обращаться к ней намеренно или иным образом. Этого можно также избежать, объявив лексическую переменную:

```
while (my $line = <STDIN>) { print $line; } # теперь закрытая
```

(Оба эти цикла while все же неявно проверяют, является ли определенным результат присвоения, поскольку my и local не влияют на то, как анализатор видит присвоение.) Дескрипторы файлов STDIN, STDOUT и STDERR являются заранее определенными и заранее открытыми. Дополнительные дескрипторы файлов могут создаваться с помощью функций open или sysopen. Подробности о них см. в документации по этим функциям в главе 29 «Функции».

В приведенных выше циклах while оператор ввода строки вычислялся в скалярном контексте, поэтому оператор возвращает отдельно каждую строку. Однако если использовать оператор в списочном контексте, возвращается список, состоящий из всех оставшихся строк, по одной строке на каждый элемент. Так можно легко создать *большое* пространство данных, поэтому применяйте эту функцию с осторожностью:

```
$one_line = <MYFILE>;   # Получить первую строку.
@all_lines = <MYFILE>;  # Получить оставшиеся строки.
```

Со списочной формой оператора ввода не связано никаких чудес while, поскольку условие цикла while всегда предоставляет скалярный контекст (как и всякий условный оператор).

Использование нулевого дескриптора файла в операторе угловых скобок представляет собой особенность: оно эмулирует режим командной строки таких типичных программ-фильтров Unix, как *sed* и *awk*. При чтении строк из <> вы чудесным образом получаете все строки файлов, перечисленных в командной строке. Если никакие файлы не указаны, вы получаете стандартное устройство ввода, поэтому вашу программу легко вставить в середину конвейера процессов.

Вот как это работает: при первом вычислении <> проверяется массив @ARGV, и если он является нулевым, то $ARGV[0] устанавливается равным «-», что при открытии дает стандартное устройство ввода. После этого массив @ARGV обрабатывается как список имен файлов. Более явно, цикл:

```
while (<>) {
    ...                     # код для каждой строки
}
```

эквивалентен следующему Perl-образному псевдокоду:

```
@ARGV = ('-') unless @ARGV;     # принять STDIN, если пуст, и только тогда
while (@ARGV) {
    $ARGV = shift @ARGV;        # каждый раз укорачивать @ARGV
    if (!open(ARGV, $ARGV)) {
        warn "Невозможно открыть $ARGV: $!\n";
        next;
    }
    while (<ARGV>) {
        ...                     # код для каждой строки
    }
}
```

за исключением того, что его не так утомительно писать, и он реально работает. Он действительно сдвигает массив @ARGV и помещает текущее имя файла в глобальную переменную $ARGV. При этом также внутренне используется особый дескриптор файла ARGV — <> служит просто синонимом более явной записи <ARGV>, являющейся магическим дескриптором файла. (Псевдокод, приведенный выше, не работает, поскольку рассматривает <ARGV> не как магический.)

Можно модифицировать @ARGV до первого <>, если в итоге этот массив будет содержать список тех имен файлов, которые действительно вам нужны. Поскольку Perl использует здесь свою обычную функцию open, то имя файла «-», если таковое встретится, считается стандартным вводом, и вам автоматически становятся доступны более скрытые возможности open (такие как откры-

тие «файла» с именем `gzip -dc < file.gz|`). Нумерация строк (`$.`) продолжается так, как если бы ввод удачно представлял собой один большой файл. (Однако посмотрите в примере для `eof` в главе 29 «Функции», как сбрасывать номера строк для каждого файла.)

Если требуется поместить в `@ARGV` собственный список файлов, то это можно сделать так:

```
# по умолчанию берется файл README, если не заданы аргументы
@ARGV = ("README") unless @ARGV;
```

При желании передать в сценарий ключи можно сделать это с помощью одного из модулей `Getopt::*` или выполнить сначала цикл такого типа:

```
while (@ARGV and $ARGV[0] =~ /^-/) {
    $_ = shift;
    last if /^- -$/;
    if (/^-D(.*)/) { $debug = $1 }
    if (/^-v/)     { $verbose++  }
    ...             # другие ключи
}

while (<>) {
    ...             # код для каждой строки
}
```

Обозначение `<>` возвратит false только один раз. Если после этого его вызвать снова, то оператор будет считать, что обрабатывается еще один список `@ARGV`, и если значения `@ARGV` не присвоены, то входные данные будут браться из `STDIN`.

Если строка в угловых скобках является скалярной переменной (например, `<$foo>`), то эта переменная содержит *косвенный* дескриптор файла, являющийся либо именем дескриптора файла, из которого нужно брать входные данные, либо ссылкой на такой дескриптор. Например:

```
$fh = \*STDIN;
$line = <$fh>;
```

или:

```
open($fh, "<data.txt");
$line = <$fh>;
```

Поиск файлов по шаблону

Вас может заинтересовать, что произойдет, если внутрь оператора ввода строки поместить что-нибудь более оригинальное. Произойдет то, что он трансформируется в другой оператор. Если строка в угловых скобках не является именем дескриптора файла или скалярной переменной (даже если там просто есть лишние пробелы), она интерпретируется как шаблон имени файла, в соответствии с которым должен быть осуществлен поиск

(«globbing»).[1] Совпадения с шаблоном отыскиваются в файлах текущего каталога (или каталога, заданного как часть шаблона), и оператор возвращает найденные имена файлов. Как и в случае оператора ввода строки, имена возвращаются по одному в скалярном контексте, или все сразу в списочном контексте. Обычно встречается последний вариант, и нередко можно видеть такие команды:

```
@files = <*.xml>;
```

Как и для других типов псевдолитералов, сначала выполняется один уровень интерполяции переменных, но для этого нельзя написать `<$foo>`, поскольку, как сказано выше, это будет воспринято как косвенный дескриптор файла. В прежних версиях Perl программисты вызывали принудительную интерпретацию строки как шаблона при помощи фигурных скобок: `<${foo}>`. В настоящее время предпочитают прямой вызов внутренней функции, как в `glob($foo)`, что, вероятно, правильно. Поэтому вместо угловых скобок вы пишете:

```
@files = glob("*.xml");
```

если с презрением относитесь к перегрузке оператора угловых скобок для этих целей. Имеете на это право.

Применяете ли вы функцию `glob` или придерживаетесь старой формы с угловыми скобками, в любом случае оператор поиска файла по шаблону осуществляет магию цикла `while`, как и оператор ввода строки, присваивая результат переменной `$_`. (Это было главной причиной перегрузки оператора угловых скобок.) Например, для того чтобы изменить права доступа ко всем файлам исходных текстов на C, можно сказать:

```
while (glob "*.c") {
    chmod 0644, $_;
}
```

что эквивалентно:

```
while (<*.c>) {
    chmod 0644, $_;
}
```

Функция `glob` в прежних версиях Perl (и даже более старых версиях Unix) была реализована как команда оболочки, что приводило к относительной дороговизне ее выполнения, а что еще хуже, она по-разному работала в раз-

---

[1] Fileglob (шаблон имени файла) не имеет никакого отношения к ранее упомянутым typeglob (шаблон имени символа), за исключением того, что в обоих символ `*` выступает в качестве группового. У символа `*`, используемого с этой целью, есть прозвище «glob». С помощью typeglob осуществляется поиск одноименных символов в таблице имен. С помощью fileglob осуществляется поиск имен файлов в каталоге по шаблону, как это происходит во многих оболочках.

личных системах. Сейчас она является встроенной, а потому более надежной и значительно более быстрой. Читайте в описании модуля File::Glob в главе 32 «Стандартные модули», как можно изменить установленный по умолчанию режим работы этого оператора: будут ли пробелы в операнде (аргументе) рассматриваться как разделители в маршруте, нужно ли расширять тильду и фигурные скобки, учитывать ли регистр, сортировать ли возвращаемые значения и т. д.

Конечно, самый короткий способ выполнения приведенной выше команды chmod и, можно утверждать, самый удобочитаемый, – это использование поиска по шаблону как списочного оператора:

```
chmod 0644, <*.c>;
```

Оператор поиска по шаблону вычисляет свой (встроенный) операнд только в начале нового списка. Все значения должны быть прочитаны, прежде чем оператор начнет работу снова. В списочном контексте это не важно, поскольку в любом случае они будут автоматически получены все. В скалярном контексте, однако, оператор при каждом вызове возвращает очередное значение или false, если они кончились. Опять же, false возвращается только один раз. Поэтому, рассчитывая, что поиск по шаблону возвратит единственное значение, гораздо лучше сказать:

```
($file) = <blurch*>;  # списочный контекст
```

чем:

```
$file = <blurch*>;    # скалярный контекст
```

поскольку первая команда возвращает все найденные имена и сбрасывает оператор поиска, в то время как вторая поочередно возвращает имена файлов, а затем false.

Пытаясь осуществить интерполяцию переменных, явно следует предпочесть оператор glob, поскольку старые обозначения могут быть спутаны с обозначениями косвенных дескрипторов файлов. Здесь становится очевидным, что граница между термами и операторами несколько расплывчата:

```
@files = <$dir/*.[ch]>;          # Действует, но лучше избегать.
@files = glob("$dir/*.[ch]");    # Вызов glob как функции.
@files = glob $some_pattern;     # Вызов glob как оператора.
```

В последнем примере мы опустили скобки, чтобы показать, что glob можно применять не только как функцию (терм), но и как *унарный* оператор; то есть префиксный оператор, принимающий один аргумент. Оператор glob служит примером *именованного унарного оператора*, одного из типов операторов, о которых мы поговорим в главе 3. Позже мы поговорим еще об операторах поиска по шаблону, которые тоже анализируются как термы, но ведут себя как операторы.

3

Унарные и бинарные операторы

В предыдущей главе речь шла о различных типах термов, которые можно использовать в выражениях, но, по правде говоря, одинокие термы довольно скучны. Термы, в большинстве своем, общественные животные. Они любят вступать в связь друг с другом. Типичный молодой терм чувствует сильную потребность солидаризироваться с другими термами и оказывать на них влияние разными способами, однако в обществе существует много видов взаимодействия и много различных уровней приверженности. В Perl эти взаимоотношения выражаются с помощью операторов.

Должна же социология приносить какую-то пользу.

С точки зрения математики операторы являются обычными функциями с особым синтаксисом. С точки зрения лингвистики операторы просто являются неправильными глаголами. Но любой лингвист скажет, что неправильные глаголы как раз чаще всего используются в языке. И это важно с точки зрения теории информации, поскольку неправильные глаголы обнаруживают тенденцию быть более краткими и более эффективными как в своем действии, так и при их распознавании.

На обычном языке это значит, что работать с операторами удобно.

Есть много сортов операторов, различия между которыми образуют *арность* (количество принимаемых операндов), *приоритет* (степень рвения, с которой они стараются отобрать эти операнды у окружающих операторов) и *ассоциативность* (предпочитаемый ими порядок работы – слева направо или справа налево – в сочетании с операторами, имеющими такой же приоритет).

По арности различают три вида операторов Perl: *унарные*, *бинарные* и *тернарные*. Унарные операторы всегда являются префиксными (за исключением постфиксных операторов инкрементирования и декрементирования).[1] Все остальные операторы – инфиксные, если не считать списковые операторы, которые могут быть префиксом для любого числа аргументов. Однако большинство воспринимает списковые операторы как обычные функции, вокруг аргументов которых допускается позабыть поставить скобки. Вот несколько примеров:

```
! $x                # унарный оператор
$x * $y            # бинарный оператор
$x ? $y : $z       # тернарный оператор
print $x, $y, $z   # списковый оператор
```

Приоритет оператора определяет прочность связывания его аргументов. Операторы с более высоким приоритетом захватывают близлежащие аргументы ранее, чем операторы с более низким приоритетом. Исходный пример взят прямо из элементарной математики, в которой умножение имеет более высокий приоритет, чем сложение:

```
2 + 3 * 4          # дает в результате 14, а не 20
```

Порядок выполнения двух операторов с одинаковым приоритетом зависит от их ассоциативности. Эти правила тоже в некоторой степени следуют соглашениям, принятым в математике:

```
2 * 3 * 4          # означает (2 * 3) * 4, левая ассоциативность
2 ** 3 ** 4        # означает 2 ** (3 ** 4), правая ассоциативность
2 != 3 != 4        # недопустимо, не ассоциативный оператор
```

В табл. 3.1 перечислены ассоциативность и арность операторов Perl, начиная с высшего приоритета и кончая низшим.

Таблица 3.1. Приоритеты операторов

Ассоциативность	Арность	Класс приоритета
Нет	0	Термы и списковые операторы (слева)
Слева	2	->
Нет	1	++ - -
Справа	2	**
Справа	1	! ~ \ и унарные + и -
Слева	2	=~ !~
Слева	2	* / % x
Слева	2	+ - .

[1] Хотя различные кавычки и скобки можно рассматривать как «циркумфиксные» операторы, определяющие границы термов.

Ассоциативность	Арность	Класс приоритета
Слева	2	`<< >>`
Справа	0,1	Именованные унарные операторы
Нет	2	`< > <= >= lt gt le ge`
Нет	2	`== != <=> eq ne cmp`
Слева	2	`&`
Слева	2	`| ^`
Слева	2	`&&`
Слева	2	`||`
Нет	2	`.....`
Справа	3	`?:`
Справа	2	`= += -= *=` и т. д.
Слева	2	`, =>`
Справа	0+	Списковые операторы (справа)
Справа	1	`not`
Слева	2	`and`
Слева	2	`or xor`

Может показаться, что уровней старшинства слишком много, чтобы их можно было запомнить. И это правда. К счастью, есть два облегчающих обстоятельства. Во-первых, уровни приоритетов определены так, что обычно соответствуют интуитивным представлениям, если только у вас все в порядке с психикой. А во-вторых, те, у кого нервы пошаливают, всегда могут поставить пару лишних скобок, чтобы избавиться от мучительного беспокойства.

Еще один полезный совет: операторы, заимствованные из C, сохраняют по отношению друг к другу принятые там приоритеты, даже когда в C для них установлено довольно странное старшинство. (Это облегчает изучение Perl тем, кто знает C и C++. И, может быть, даже Java.)

В последующих разделах будет рассказано об этих операторах в порядке старшинства. За очень малым исключением все они действуют только над скалярными величинами, а не над списками. Мы укажем исключения, когда дойдем до соответствующих мест.

Хотя ссылки и являются скалярными величинами, но применение большинства этих операторов к ссылкам не очень разумно, поскольку числовое значение ссылки имеет смысл только для самого Perl. Однако если ссылка указывает на объект класса, допускающего перегрузку, то к такому объекту эти операторы можно применять, а если в классе определена перегрузка некоторого оператора, то она определяет действие над объектом при применении к нему данного оператора. Таким способом, например, в Perl реализованы комплексные числа. Более подробно перегрузка рассмотрена в главе 13 «Перегрузка».

Термы и списковые операторы (слева)

Любой *терм* имеет в Perl высший приоритет. В число термов входят переменные, операторы кавычек и подобные им, большинство выражений в круглых, квадратных или фигурных скобках и любые функции, аргументы которых заключены в скобки. Фактически функции в этом смысле отсутствуют, есть только списковые и унарные операторы, ведущие себя как функции, поскольку их аргументы заключены в круглые скобки. Тем не менее, глава 29 называется «Функции».

Теперь слушайте внимательно. Есть пара очень важных правил, существенно упрощающих дело, но приводящих иногда к неожиданным результатам, если не проявлять осмотрительность. Если за списковым оператором (например, `print`) или любым именованным унарным оператором (например, `chdir`) в качестве следующей лексемы следует левая круглая скобка (не считая пробельных символов), то оператор и аргументы в скобках получают высший приоритет, как если бы это был обычный вызов функции. Правило следующее: если это *похоже* на вызов функции, то это и *есть* вызов функции. Можно сделать так, чтобы он не выглядел как функция, поместив перед скобками унарный плюс, который, семантически говоря, не делает абсолютно ничего – даже не принуждает аргумент к переводу в число.

Например, поскольку `||` имеет более низкий приоритет, чем `chdir`, мы получим:

```
chdir $foo    || die;       # (chdir $foo) || die
chdir($foo)   || die;       # (chdir $foo) || die
chdir ($foo)  || die;       # (chdir $foo) || die
chdir +($foo) || die;       # (chdir $foo) || die
```

но т. к. `*` имеет более высокий приоритет, чем `chdir`, получится:

```
chdir $foo * 20;            # chdir ($foo * 20)
chdir($foo) * 20;           # (chdir $foo) * 20
chdir ($foo) * 20;          # (chdir $foo) * 20
chdir +($foo) * 20;         # chdir ($foo * 20)
```

Аналогично для любого числового оператора, который является именованным унарным оператором, например оператора `rand`:

```
rand 10 * 20;               # rand (10 * 20)
rand(10) * 20;              # (rand 10) * 20
rand (10) * 20;             # (rand 10) * 20
rand +(10) * 20;            # rand (10 * 20)
```

В отсутствие круглых скобок приоритет списковых операторов, таких как `print`, `sort` или `chmod`, оказывается либо очень высоким, либо очень низким, в зависимости от того, на какую сторону оператора смотреть – левую или правую. (Это и есть «слева», выведенное в заголовок этого раздела.) Например, в:

```
@ary = (1, 3, sort 4, 2);
print @ary;         # выводит 1324
```

запятые справа от sort вычисляются перед sort, но запятые слева вычисляются после него. Иными словами, списковый оператор стремится проглотить все аргументы, которые следуют после него, а затем действовать как простой терм в предшествующем выражении. Но со скобками нужно по-прежнему обращаться осторожно:

```
# Эти операторы выполняют exit раньше, чем print:
print($foo, exit);  # Очевидно, это не то, что нужно.
print $foo, exit;   # Как и это.

# Эти операторы выполняют print раньше, чем exit:
(print $foo), exit; # Это то, что нужно.
print($foo), exit;  # И это.
print ($foo), exit; # И даже это.
```

Чаще всего обжигаются на использовании скобок для группировки математических аргументов, забывая при этом, что скобки применяются также для группировки аргументов функций:

```
print ($foo & 255) + 1, "\n";   # выводит ($foo & 255)
```

Это выполняется не так, как может показаться с первого взгляда. К счастью, ошибки такого вида обычно вызывают вывод предупредительных сообщений типа «Useless use of addition in a void context» (бесполезное сложение в пустом контексте), если вывод предупредительных сообщений включен.

Конструкции do {} и eval {} тоже выделяются анализатором как термы, как и вызовы подпрограмм и методов, конструкторы анонимных массивов и хешей [] и {}, а также анонимных подпрограмм sub {}.

Оператор «стрелка»

Точно так же, как в C и C++, двоичный оператор -> является инфиксным оператором разыменования. Если правая часть представляет собой индекс массива [...], индекс хеша {...} или список аргументов подпрограммы (...), то левая часть должна быть ссылкой (жесткой или символической) на массив, хеш и подпрограмму, соответственно. В контексте l-значения (допускающем присваивание), когда левая часть не является ссылкой, она должна быть неким адресом, по которому может храниться жесткая ссылка, и в этом случае происходит самооживление (autovivification) такой ссылки. Более подробно об этом (с некоторыми предупреждениями относительно случайного самоооживления) рассказано в главе 8 «Ссылки».

```
$aref->[42]              # разыменование массива
$href->{"corned beef"}   # разыменование хеша
$sref->(1,2,3)           # разыменование подпрограммы
```

В иных случаях это того или иного рода вызов метода. Правая часть должна быть именем метода (или простой скалярной переменной, содержащей имя

метода), а левая часть должна вычисляться как объект (связанная с объектом ссылка) или имя класса (т. е. имя пакета):

```
$yogi = Bear->new("Yogi");      # вызов метода класса
$yogi->swipe($picnic);         # вызов метода объекта
```

Имя метода может быть квалифицировано именем пакета для указания класса, с которого нужно начать поиск метода, либо специальным именем пакета `SUPER::`, указывающим, что поиск должен быть начат с родительского класса. См. главу 12 «Объекты».

Автоинкрементирование и автодекрементирование

Операторы ++ и -- работают так же, как в C. Это значит, что, будучи помещены перед переменной, они инкрементируют или декрементируют переменную перед тем, как возвратить ее значение, а будучи помещены после нее, они инкрементируют или декрементируют переменную после того, как возвращено ее значение. Например, `$a++` инкрементирует значение скалярной переменной `$a`, возвращая ее значение *перед* выполнением инкрементирования. Аналогично `--$b{(/(\w+)/)[0]}` декрементирует элемент хеша `%b` с индексом, равным первому «слову» в установленной по умолчанию для поиска переменной (`$_`) и возвращает значение, полученное *после* декрементирования.[1]

В оператор автоинкрементирования встроено еще одно «магическое» свойство. При инкрементировании переменной, являющейся числом или когда-либо появлявшейся в числовом контексте, мы получаем обычное инкрементирование. Если, однако, переменная после присваивания ей значения использовалась только в строковых контекстах и имеет значение, отличное от нулевой строки и соответствующее шаблону `/^[a-zA-Z]*[0-9]*$/`, то выполняется инкрементирование строки с сохранением символов в этом диапазоне и переносом:

```
print ++($foo = '99');      # выводит '100'
print ++($foo = 'a9');      # выводит 'b0'
print ++($foo = 'Az');      # выводит 'Ba'
print ++($foo = 'zz');      # выводит 'aaa'
```

[1] На самом деле, это не совсем честно. Мы просто хотели проверить вашу внимательность. Вот как работает это выражение. Сначала поиск по шаблону находит первое слово в `$_`, используя регулярное выражение `\w+`. Благодаря окружающим его скобкам найденное значение возвращается как список из одного элемента, поскольку поиск по шаблону выполняется в списковом контексте. Списковый контекст предоставляется оператором среза списка, `(...)[0]`, который возвращает первый (и единственный) элемент списка. Это значение используется как ключ для хеша, элемент хеша (значение) декрементируется и возвращается. В общем, сталкиваясь со сложным выражением, анализируйте его в направлении «изнутри – наружу», чтобы установить порядок, в котором происходят события.

На момент написания книги магическое автоинкрементирование не было распространено на буквы и цифры Unicode, но в будущем это может быть сделано.

Однако оператор автодекрементирования магическим не является, и мы не планируем сделать его таковым.

Возведение в степень

Бинарный оператор `**` осуществляет возведение в степень. Обратите внимание, что он связывает аргументы даже сильнее, чем унарный минус, поэтому `-2**4` равно `-(2**4)`, а не `(-2)**4`. Этот оператор реализован с помощью С-функции *pow*(3), использующей (внутренне) числа с плавающей запятой. Расчет выполняется с использованием логарифмов, поэтому можно возводить и в дробные степени, но иногда получаемые результаты не столь точны, какими могли бы быть при обычном умножении.

Идеографические унарные операторы

Большинству унарных операторов присвоены имена (см. раздел «Именованные унарные операторы и операторы проверки файлов» далее в этой главе), но некоторые операторы считаются настолько важными, что заслужили собственное особое символическое представление. Все эти операторы оказываются имеющими отношение к отрицанию. Вините математиков.

Унарный `!` осуществляет логическое отрицание, то есть «не». Вариант логического отрицания с более низким приоритетом – `not`. Значением отрицаемого операнда становится «истина», или true, (1), если операнд есть «ложь», или false, (число 0, строка `"0"`, нулевая строка или неопределенное значение), и «ложь», или false, (`""`), если операнд есть «истина».

Унарный – осуществляет арифметическое отрицание, если операнд является числом. Если операнд представляет собой идентификатор, то возвращается строка, состоящая из знака «минус», конкатенированного с идентификатором. В иных случаях, если строка начинается со знака («плюс» или «минус»), возвращается строка, начинающаяся с противоположного знака. Одним из результатов этих правил является то, что `-bareword` эквивалентно `"-bareword"`. Это очень удобно для программирующих на Tk.

Унарный `~` осуществляет поразрядное отрицание, т. е. дополнение до 1. Из определения следует, что этот оператор не вполне переносим, поскольку связан с размером машинного слова. Например, на 32-разрядной машине `~123` равно `4294967172`, а на 64-разрядной равно `18446744073709551492`. Но об этом читатель уже знает.

О чем он, возможно, не знает, так это о том, что если аргументом `~` оказывается не число, а строка, то возвращается строка той же длины, но все разряды ее являются дополнениями до 1. Это быстрый способ перекинуть за одно обращение большое количество битов, к тому же это переносимый способ,

поскольку результат не зависит от размера машинного слова. Далее мы еще расскажем о поразрядных логических операторах, имеющих варианты, ориентированные на строки.

Унарный + не оказывает никакого семантического действия – даже на строки. Его синтаксическая польза состоит в отделении имени функции от выражения в скобках, которое иначе интерпретировалось бы как полный список аргументов функции. (См. примеры раздела «Термы и списковые операторы» данной главы.) Можно представить себе это и так, что + отрицает действие скобок по превращению префиксных операторов в функции.

Унарный оператор \ создает ссылку на то, что следует за ним. Применение его к списку создает список ссылок. Подробности можно найти в разделе «Оператор обратной косой черты» главы 8 «Ссылки». Не смешивайте действие этого оператора с действием обратной косой черты внутри строки, хотя в обоих случаях передается некоторая идея отрицания, для того чтобы защитить последующий объект от интерпретации. Это сходство не является полностью случайным.

Операторы связывания

Бинарный оператор =~ связывает строковое выражение с поиском по шаблону, подстановкой или транслитерацией (вольно называемой трансляцией). Если не использовать этот оператор, то поиск или замена осуществляются в строке, содержащейся в переменной $_ (переменной по умолчанию). Строка, подлежащая связыванию, помещается в левую часть, а сам оператор – справа от нее. Возвращаемое значение указывает на успех или неудачу оператора справа, поскольку оператор связывания, в сущности, никаких самостоятельных действий не выполняет.

Если правый аргумент представляет собой выражение, а не операторо поиска по шаблону, замены или транслитерации, то на этапе выполнения это выражение будет интерпретироваться как шаблон, по которому должен быть проведен поиск. Например, `$_ =~ $pat` эквивалентно `$_ =~ /$pat/`. Это менее эффективно, чем явный поиск, поскольку шаблон должен проверяться и, возможно, перекомпилироваться при каждом вычислении выражения. Избежать повторной компиляции можно путем предварительной компиляции исходного шаблона с использованием оператора `qr//` (заключения в кавычки регулярного выражения).

Бинарный оператор !~ аналогичен =~, с тем отличием, что возвращается логическое отрицание результата. Следующие два выражения действуют одинаково:

```
$string !~ /pattern/
not $string =~ /pattern/
```

Мы сказали, что возвращаемое значение сигнализирует об успешности операции, но есть разные типы успеха. Подстановка возвращает число найден-

ных совпадений, как и транслитерация. (На практике оператор транслитерации часто применяют для подсчета символов.) Поскольку любой ненулевой результат является истиной, то все работает. Наиболее впечатляющий пример истинного значения дает присваивание шаблона списку: в списковом контексте поиск по шаблону может возвращать подстроки, соответствующие круглым скобкам, имеющимся в шаблоне. И снова, согласно правилам присваивания списку сама операция присваивания возвращает истину, если что-либо было найдено и присвоено, и ложь в противном случае. Поэтому иногда можно встретить код такого вида:

```
if ( ($k,$v) = $string =~ m/(\w+)=(\w*)/ ) {
    print "КЛЮЧ $k ЗНАЧЕНИЕ $v\n";
}
```

Рассмотрим этот пример по частям. Оператор `=~` имеет приоритет над `=`, поэтому сначала происходит `=~`. Оператор `=~` связывает `$string` с поиском по шаблону в правой части, который ищет в строке подстроку, похожую на `КЛЮЧ=ЗНАЧЕНИЕ`. Он находится в списковом контексте, поскольку расположен в правой части присваивания списку. Если поиск успешен, то он возвращает список, который нужно присвоить `$k` и `$v`. Само присваивание списку находится в скалярном контексте, поэтому оно возвращает `2` — число значений в правой части присваивания. А `2` оказывается истиной, поскольку наш скалярный контекст является также и булевым. Если поиск неудачен, то значения не присваиваются и возвращается 0, который является ложью.

Дополнительно о работе с шаблонами см. главу 5 «Поиск по шаблону».

Мультипликативные операторы

В Perl имеются C-подобные операторы `*` (умножение), `/` (деление) и `%` (взятие по модулю). Операторы `*` и `/` действуют точно так, как можно предположить, умножая или деля два свои операнда. Деление осуществляется с плавающей точкой, если не используется модуль прагмы `integer`.

Оператор `%` преобразует свои операнды в целые числа, а затем находит остаток от деления целых чисел. (Однако при необходимости он осуществляет это деление с плавающей точкой, поэтому на большинстве 32-разрядных машин операнды могут иметь размер до 15 цифр.) Допустим, что имена этих двух операндов суть `$a` и `$b`. Если `$b` положительно, то результатом `$a % $b` будет `$a` минус наибольшее кратное `$b`, не превышающее `$a` (вследствие чего результат всегда будет находиться в диапазоне `0 .. $b -1`). Если `$b` отрицательно, то результатом `$a % $b` будет `$a` минус наименьшее кратное `$b`, которое не меньше `$a` (вследствие чего результат всегда будет находиться в диапазоне `$b +1 .. 0`).

Если в области видимости имеется директива `use integer`, то `%` предоставляет прямой доступ к оператору взятия по модулю в том виде, в каком он реализован в вашем компиляторе C. Этот оператор не вполне корректно определен для отрицательных операндов, но выполняется быстрее.

Бинарный оператор x является оператором повторения. Фактически в нем два разных оператора. В скалярном контексте он возвращает конкатенированную строку, состоящую из левого операнда, повторенного число раз, заданное правым операндом. (Для обратной совместимости он делает это и в списковом контексте, если левый аргумент не заключен в круглые скобки.)

```
print '-' x 80;                          # выводит строку дефисов
print "\t" x ($tab/8), ' ' x ($tab%8);   # замена пробелов табуляцией
```

В списковом контексте, если левый операнд заключен в круглые скобки, то x действует как репликатор списка, а не репликатор строки. Это удобно для инициализации массива неопределенной длины одинаковыми значениями:

```
@ones = (1) x 80;          # список из 80 единиц
@ones = (5) x @ones;       # установить все элементы равными 5
```

Аналогично оператор x можно использовать для инициализации срезов массивов и хешей:

```
@keys = qw(perls before swine);
@hash{@keys} = ("") x @keys;
```

Если вам это непонятно, то обратите внимание, что @keys выступает как список в левой части присваивания и как скалярная величина (возвращающая размер массива) в правой части сравнения. Предыдущий пример оказывает на %hash такое же воздействие, как:

```
$hash{perls}  = "";
$hash{before} = "";
$hash{swine}  = "";
```

Аддитивные операторы

Как ни странно, в Perl есть и обычные операторы + (сложение) и - (вычитание). Оба оператора при необходимости преобразуют свои аргументы из строк в числа и возвращают числовой результат.

Кроме того, в Perl имеется оператор . для конкатенации строк. Например:

```
$almost = "Fred" . "Flintstone";    # возвращает FredFlintstone
```

Заметьте, что Perl не помещает пробел между конкатенируемыми строками. Если пробел необходим или нужно конкатенировать более двух строк, можно использовать оператор join, описываемый в главе 29 «Функции». Чаще всего, однако, конкатенацию выполняют неявным образом, заключая строку в двойные кавычки:

```
$fullname = "$firstname $lastname";
```

Операторы сдвига

Операторы поразрядного сдвига (<< и >>) возвращают значение левого аргумента, сдвинутое влево (<<) или вправо (>>) на число разрядов, заданное правым аргументом. Аргументы должны быть целыми числами. Например:

```
1 << 4;      # возвращает 16
32 >> 4;     # возвращает 2
```

Пользуйтесь, однако, этими операторами с осторожностью. Результат применения их к большим (или отрицательным) числам может различаться в зависимости от количества разрядов, используемых в вашей машине для представления целых чисел.

Именованные унарные операторы и операторы проверки файлов

Некоторые из «функций», описываемых в главе 29 «Функции», являются в действительности унарными операторами. В табл. 3.2 перечислены все именованные унарные операторы.

Таблица 3.2. Именованные унарные операторы

-X (проверка файлов)	gethostbyname	localtime	return
alarm	getnetbyname	lock	rmdir
caller	getpgrp	log	scalar
chdir	getprotobyname	lstat	sin
chroot	glob	my	sleep
cos	gmtime	oct	sqrt
defined	goto	ord	srand
delete	hex	quotemeta	stat
do	int	rand	uc
eval	lc	readlink	ucfirst
exists	lcfirst	ref	umask
exit	length	require	undef

Унарные операторы имеют более высокий приоритет, чем некоторые бинарные. Например:

```
sleep 4 | 3;
```

«засыпает» не на 7 секунд, а на 4, а затем берет результат, возвращаемый `sleep` (обычно 0), и выполняет его поразрядное ИЛИ с 3, как если бы скобки были расставлены так:

```
(sleep 4) | 3;
```

Сравните с:

```
print 4 | 3;
```

Этот оператор выполняет логическое ИЛИ 4 и 3 (7 в данном случае), *перед* выводом, как если бы стояли скобки:

```
print (4 | 3);
```

Дело в том, что `print` представляет собой списковый оператор, а не простой унарный. Если запомнить, какие операторы являются списковыми, то несложно отличить унарный оператор от спискового. При возникновении сомнений всегда можно воспользоваться круглыми скобками, чтобы превратить унарный оператор в функцию. Помните: то, что выглядит как функция, функцией и является.

Еще одно любопытное обстоятельство, связанное с унарными операторами, заключается в том, что многие из них по умолчанию работают с `$_`, если не передать им никакого аргумента. Однако если аргумент опущен, но лексема, следующая за унарным оператором, похожа на начало аргумента, Perl придет в замешательство, поскольку он ждет появления терма. Когда анализатор лексем Perl получает один из символов, перечисленных в табл. 3.3, он возвращает различные типы лексем в зависимости от того, что он рассчитывает получить – терм или оператор.

Таблица 3.3. Двусмысленные символы

Символ	Оператор	Терм
+	Сложение	Унарный плюс
-	Вычитание	Унарный минус
*	Умножение	`*typeglob`
/	Деление	`/шаблон/`
<	Меньше, сдвиг влево	`<HANDLE>, <<END`
.	Конкатенация	`.3333`
?	`?:`	`?шаблон?`
%	Взятие по модулю	`%assoc`
&	`&, &&`	`&subroutine`

Типичная глупая ошибка выглядит так:

```
next if length < 80;
```

В этом случае < кажется анализатору началом символа ввода <> (термом), а не оператором «меньше», который вам нужен. Исправить эту ситуацию в Perl и одновременно сохранить его патологическую эклектичность решительно невозможно. Если вы невероятно ленивы и не можете заставить себя напечатать два символа $_, воспользуйтесь одним из следующих способов:

```
next if length() < 80;
next if (length) < 80;
next if 80 > length;
next unless length >= 80;
```

Если ожидается терм, то знак «минус», за которым следует одна буква, всегда рассматривается как оператор *проверки файла*. Оператор проверки файла представляет собой унарный оператор, который принимает один аргумент, являющийся именем файла или его дескриптором, и проверяет, обладает ли этот файл некоторым свойством. Если аргумент опущен, оператор проверяет $_, за исключением оператора -t, который проверяет STDIN. Если не указано иное, возвращаемым значением является 1 при истинности и "" при ложности либо неопределенное значение, если файл не существует или недоступен по каким-то другим причинам. Реализованные в настоящее время операторы проверки файлов перечислены в табл. 3.4.

Таблица 3.4. Операторы проверки файлов

Оператор	Значение
-r	Файл доступен для чтения текущему пользователю или группе (effective UID/GID)
-w	Файл доступен для записи текущему пользователю или группе (effective UID/GID)
-x	Файл доступен для выполнения текущему пользователю или группе (effective UID/GID)
-o	Файл принадлежит текущему пользователю (effective UID)
-R	Файл доступен для чтения реальному пользователю или группе (real UID/GID)
-W	Файл доступен для записи реальному пользователю или группе (real UID/GID)
-X	Файл доступен для выполнения реальному пользователю или группе (real UID/GID)
-O	Файл принадлежит реальному пользователю (real UID)
-e	Файл существует
-z	Файл имеет нулевую длину
-s	Файл имеет ненулевую длину (возвращает размер)
-f	Обычный файл
-d	Каталог
-l	Файл является символической ссылкой

Таблица 3.4 (продолжение)

Оператор	Значение
-p	Файл является именованным каналом (FIFO)
-S	Файл является сокетом
-b	Блочный файл
-c	Символьный файл
-t	Дескриптор файла открыт на текущем терминале
-u	Для файла установлен бит setuid
-g	Для файла установлен бит setgid
-k	Для файла установлен «липкий бит» (sticky bit)
-T	Файл является текстовым
-B	Файл является двоичным (в противоположность -T)
-M	Время, прошедшее от последней модификации файла до запуска сценария, в днях
-A	Время, прошедшее от последнего обращения к файлу до запуска сценария, в днях
-C	Время, прошедшее от последнего изменения индексного дескриптора до запуска сценария, в днях

Обратите внимание, что -s/a/b/ не производит подстановки с отрицанием. Однако -exp($foo) работает, как должно: только одиночные буквы после минуса трактуются как проверка файла.

Интерпретация операторов прав доступа к файлам -r, -R, -w, -W, -x и -X основывается исключительно на свойствах файла и пользовательском и групповом идентификаторах пользователя. Могут быть и другие причины, по которым не удается читать, записывать или выполнять файл, например, списки управления доступом файловой системы Andrew File System (AFS).[1] Заметьте также, что для суперпользователя -r, -R, -w и -W всегда возвращают 1, а -x и -X возвращают 1, если установлен хоть один бит исполнения. Поэтому в сценариях, выполняемых суперпользователем, может потребоваться выполнить stat, чтобы определить действительный режим файла, либо временно установить другой идентификатор пользователя.

Остальным операторам проверки файлов безразлично, кто вы такой. Проверку «регулярности» файлов может выполнять любой:

```
while (<>) {
    chomp;
    next unless -f $_;      # пропустить "особые" файлы
    ...
}
```

[1] Однако можно переопределить встроенную семантику с помощью директивы use filetest. См. главу 31 «Модули прагм».

Ключи -T и -B работают следующим образом. Участок файла примерно в объеме первого блока исследуется на наличие необычных символов, таких как коды управления или байты с установленным старшим битом (не похожие на UTF-8). Если необычными оказываются более одной трети байтов, то файл считается двоичным, в противном случае – текстовым. Кроме того, двоичным считается любой файл, в первом блоке которого встретился символ ASCII NUL (\0). Если -T или -B применяется к дескриптору файла, то исследуется не первый блок файла, а текущий буфер ввода (стандартный I/O или «stdio»). Оба оператора, -T и -B, возвращают истину для пустого файла или файла, указатель которого находится в EOF – конце файла, если проверяется дескриптор. Поскольку Perl должен прочесть файл, чтобы выполнить проверку -T, применение последней со специальными файлами, которые могут вызвать зависание или доставить другие неприятности, нежелательно. Поэтому чаще всего нужно сначала выполнить проверку с ключом -f, например:

```
next unless -f $file && -T $file;
```

Если какой-либо из проверок файлов (или операторам stat или lstat) передается специальный дескриптор файла, состоящий из одного символа подчеркивания, то используется структура *stat* от предыдущей проверки файла (или оператора stat), что позволяет избежать нового системного вызова. (Это не действует для оператора -t, и нужно помнить, что lstat и -l оставляют в структуре *stat* значения, относящиеся к символической ссылке, а не к реальному файлу. Аналогично, -l _ всегда возвращает «ложь» после обычного stat.)

Вот несколько примеров:

```
print "Доступен для обработки .\n"    if -r $a || -w _ || -x _;

stat($filename);
print "Доступен для чтения\n"   if -r _;
print "Доступен для записи\n"   if -w _;
print "Исполняемый \n" if -x _;
print "Установлен бит setuid\n"     if -u _;
print "Установлен бит setgid\n"     if -g _;
print "Текстовый\n"       if -T _;
print "Двоичный\n"     if -B _;
```

Время существования файла операторами -M, -A и -C возвращается в днях (считая дробную часть суток) на момент начала выполнения сценария. Это время хранится в особой переменной $^T ($BASETIME). Поэтому, если файл был изменен после начала выполнения сценария, возвращается отрицательное время. Обратите внимание, что значения времени в большинстве случаев (в среднем, 86399 из 86400) являются дробными, поэтому проверка равенства целому числу без использования функции int обычно оказывается бесполезной. Примеры:

```
next unless -M $file > .5;       # файлы старше 12 часов
&newfile if -M $file < 0;        # файл новее, чем процесс
```

```
&mailwarning if int(-A) == 90;  # обращение к файлу ($_) ????? ????? ровно 90 дней
назад
```

Чтобы установить время запуска сценария равным текущему времени, достаточно выполнить:

```
$^T = time;
```

Операторы сравнения

В Perl имеется два класса операторов сравнения. Один класс выполняет операции сравнения со строковыми величинами, другой с числовыми, как показано в табл. 3.5.

Таблица 3.5. Операторы сравнения

Числа	Строки	Значение
>	gt	Больше
>=	ge	Больше или равно
<	lt	Меньше
<=	le	Меньше или равно

Эти операторы возвращают 1 в качестве истины и "" – как ложь. Заметьте себе, что операторы сравнения неассоциативны, что означает синтаксическую ошибку в выражении $a < $b < $c.

В отсутствие объявлений национальных настроек сравнение строк основывается на схеме упорядочения ASCII/Unicode, при этом, в отличие от некоторых языков программирования, концевые пробелы участвуют в сравнении. При наличии объявления национальных установок используется определенная в нем схема упорядочения. (Механизмы сравнения, основывающиеся на национальных установках, не всегда могут хорошо взаимодействовать с механизмами сравнения Unicode, находящимися в настоящее время в стадии разработки.)

Операторы равенства

Операторы равенства, перечисленные в табл. 3.6, во многом похожи на операторы сравнения.

Таблица 3.6. Операторы равенства

Числа	Строки	Значение
==	Eq	Равно
!=	Ne	Не равно
<=>	Cmp	Сравнение, результат со знаком

Операторы проверки равенства и неравенства возвращают `1` в качестве значения «истина» и `""` в качестве значения «ложь» (так же, как операторы сравнения). Операторы `<=>` и `cmp` возвращают `-1`, если левый операнд меньше правого, `0` при их равенстве и `+1`, если левый операнд больше правого. Хотя операторы равенства кажутся аналогичными операторам сравнения, они имеют более низкий приоритет, поэтому запись `$a < $b <=> $c < $d` синтаксически допустима.

По причинам, очевидным для каждого видевшего фильм «Звездные войны», оператор `<=>` называют также оператором «космический корабль» (spaceship).

Операторы поразрядного действия

Как и в C, в Perl есть поразрядные операторы И, ИЛИ и исключающее ИЛИ (XOR): `&`, `|`, и `^`. Взяв на себя труд изучить таблицу в начале этой главы, читатель мог обнаружить, что поразрядное И имеет более высокий приоритет, чем остальные поразрядные операторы, но мы смошенничали и объединили их в нашем изложении.

Эти операторы по-разному действуют на числа и строки. (Один из немногих случаев, когда Perl учитывает разницу между ними.) Если оба операнда – числа (или использовались как числа), они преобразуются в целые числа и поразрядная операция производится над двумя числами. Эти числа имеют гарантированную длину 32 разряда, но на некоторых машинах это может быть 64 разряда. Дело в том, что существует произвольное ограничение, налагаемое архитектурой машины.

Если оба операнда суть строки (и не использовались как числа с момента присваивания им значений), то операторы выполняют поразрядные операции над соответствующими разрядами двух строк. В этом случае не действует произвольное ограничение, поскольку для размера строк произвольного ограничения нет. Если одна строка длиннее, чем другая, считается, что более короткая строка имеет в конце достаточное число нулевых разрядов для компенсирования разницы.

Например, если выполнить И над двумя строками:

```
"123.45" & "234.56"
```

то получится новая строка:

```
"020.44"
```

Но если выполнить И над строкой и числом:

```
"123.45" & 234.56
```

то строка сначала преобразуется в число, что дает:

```
123.45 & 234.56
```

Затем числа преобразуются в целые:

```
123 & 234
```

и в результате получается 106. Обратите внимание, что все битовые строки истинны (если только не равны строке "0"). Это значит, что если требуется определить, оказался ли какой-то байт ненулевым, то вместо:

```
if ( "fred" & "\1\2\3\4" ) { ... }
```

нужно записать:

```
if ( ("fred" & "\1\2\3\4") =~ /[^\0]/ ) { ... }
```

Логические операторы в стиле C (короткого действия)

Как и в C, в Perl имеются операторы && (логическое И) и || (логическое ИЛИ). Они выполняют вычисления слева направо (при этом && имеет несколько больший приоритет, чем ||), проверяя утверждения на истинность. Эти операторы известны как операторы короткого действия, поскольку определяют истинность утверждения, вычисляя возможно меньшее число операндов. Например, если левый операнд оператора && имеет значение «ложь», то правый операнд не вычисляется вообще, поскольку вне зависимости от его значения результат имеет значение «ложь».

Пример	Название	Результат
$a && $b	И	$a, если $a ложно, в противном случае $b
$a || $b	ИЛИ	$a, если $a истинно, в противном случае $b

Это короткое действие не только экономит время, но часто используется для управления порядком вычислений. Например, в программах Perl часто встречается идиома:

```
open(FILE, "некий_файл") || die "Невозможно открыть некий_файл: $!\n";
```

В данном случае Perl сначала вычисляет функцию open. Если ее значение истинно (т. е. *некий_файл* был успешно открыт), то выполнение функции die не является обязательным и пропускается. Можно прочесть это буквально: «Открыть некий файл или умереть!»

Операторы && и || отличаются от имеющихся в C тем, что возвращают не 0 или 1, а последнее вычисленное значение. В случае || это приводит к тому приятному результату, что можно выбрать первое из ряда скалярных значений, которое окажется истинным. Поэтому весьма переносимым способом найти исходный каталог пользователя может быть такой:

```
$home = $ENV{HOME}
     || $ENV{LOGDIR}
```

```
|| (getpwuid($<))[7]
|| die "Да ты бездомный!\n";
```

С другой стороны, поскольку левый аргумент всегда вычисляется в скалярном контексте, нельзя использовать || при выборе для присваивания одного из двух составных объектов:

```
@a = @b || @c;              # Получится не то, что нужно, поскольку
@a = scalar(@b) || @c;      # в действительности будет вот что.
@a = @b ? @b : @c;          # А это работает прекрасно.
```

В Perl есть также операторы с более низким приоритетом, and и or, которые некоторым кажутся более удобочитаемыми и не принуждают использовать скобки в списковых операторах. Они тоже имеют короткое действие. Полный список можно найти в табл. 1.1.

Оператор диапазона

Оператор диапазона .. представляет собой, в сущности, два разных оператора в зависимости от контекста.

В скалярном контексте .. возвращает булево значение. Оператор диапазона, подобно электронному триггеру, имеет два устойчивых состояния и эмулирует оператор диапазона строки (запятая) из *sed*, *awk* и различных редакторов. Каждый скалярный оператор .. сохраняет собственное булево состояние. Оно имеет значение «ложь», если левый операнд имеет значение «ложь». Если левый операнд имеет значение «истина», оператор диапазона остается истинным, пока правый операнд не получит значение «истина», *после* чего оператор диапазона снова получает значение «ложь». Оператор не получит значения «ложь», пока не будет вычислен в очередной раз. Он может проверить правый операнд и стать ложным в том же вычислении, когда стал истинным (так ведет себя оператор диапазона *awk*), но все равно один раз возвратит значение «истина». Если вы не хотите, чтобы правый операнд проверялся до следующего вычисления (а так действует оператор диапазона *sed*), нужно вместо двух точек использовать три (...). Как для .., так и для ... правый операнд не проверяется, пока оператор находится в состоянии «ложь», а левый операнд не проверяется, пока оператор находится в состоянии «истина».

Возвращаемое значение является либо пустой строкой для значения «ложь», либо порядковым номером (начинающимся с 1) для значения «истина». Последовательный номер сбрасывается для каждого встретившегося диапазона. К последнему номеру в диапазоне добавляется строка E0, которая не влияет на числовое значение, но дает значение, в котором можно осуществлять поиск при желании исключить конечную точку. Исключить начальную точку можно, ожидая, когда последовательный номер станет больше 1. Если оба операнда скалярного .. представляют собой числовые литералы, то операнды неявно сравниваются с переменной $., которая содержит номер строки входного файла. Примеры:

```
if (101 .. 200) { print; }  # вывести вторую сотню строк
next line if (1 .. /^$/);   # пропустить заголовочные строки сообщения
s/^/> / if (/^$/ .. eof()); # заключить в кавычки тело сообщения
```

В списковом контексте `..` возвращает список значений, считая от левого до правого операндов. Это удобно при записи циклов `for (1..10)` и операций среза над массивами:

```
for (101 .. 200) { print; }          # выводит 101102...199200
@foo = @foo[0 .. $#foo];             # дорогостоящий пустой код операции
@foo = @foo[ -5 .. -1];              # срез из последних 5 элементов
```

Если левое значение больше правого, возвращается пустой список. (Для создания списка, следующего в обратном порядке, используется оператор `reverse`.)

Если операнды являются строками, оператор диапазона применяет алгоритм магического автоинкрементирования, обсуждавшийся ранее.[1] Поэтому можно сказать:

```
@alphabet = ('A' .. 'Z');
```

и получить все буквы (английского) алфавита, либо:

```
$hexdigit = (0 .. 9, 'a' .. 'f')[$num & 15];
```

и получить шестнадцатеричную цифру, либо:

```
@z2 = ('01' .. '31');  print $z2[$mday];
```

и получить дату с ведущими нулями. Можно также сказать:

```
@combos = ('aa' .. 'zz');
```

и получить все комбинации из двух букв нижнего регистра. Однако остерегайтесь таких выражений:

```
@bigcombos = ('aaaaaa' .. 'zzzzzz');
```

поскольку для них требуется большой объем памяти. Если точно, то потребуется память для записи 308 915 776 скаляров. Будем надеяться, что для свопинга выделен большой раздел. Возможно, итеративный подход лучше.

Условный оператор

Как и в C, `?:` является единственным тернарным оператором. Он часто называется условным оператором, поскольку во многом работает сходно с if-then-

[1] Если конечное значение не входит в последовательность, которую создает магическое инкрементирование, последовательность продолжается, пока ее очередное значение не станет длиннее указанного конечного значения.

else, за исключением того, что, поскольку это выражение, а не команда, его можно безопасно встраивать в другие выражения и вызовы функций. Поскольку это тернарный оператор, две его части разделяют три выражения:

```
COND ? THEN : ELSE
```

Если условие *COND* является истинным, то вычисляется только выражение *THEN*, значение которого становится значением выражения в целом. В противном случае вычисляется только выражение *ELSE*, значение которого становится значением выражения в целом.

Скалярный или списковый контекст распространяется далее на второй или третий аргумент в зависимости от того, который из них выбран. (Первый аргумент всегда находится в скалярном контексте, поскольку этот оператор является условным.)

```
$a = $ok ? $b : $c;  # получить скаляр
@a = $ok ? @b : @c;  # получить массив
$a = $ok ? @b : @c;  # получить счетчик числа элементов массива
```

Часто можно видеть условный оператор встроенным в список значений для форматирования посредством `printf`, поскольку никому не хочется повторять целую команду для того, чтобы переключаться между двумя связанными значениями.

```
printf "I have %d camel%s.\n",
               $n,     $n == 1 ? "" : "s";
```

Удобно, что приоритет `?:` выше, чем у запятой, но ниже, чем у большинства операторов, используемых внутри (`==` в данном примере), поэтому обычно не приходится расставлять скобки. Но для ясности можно, при желании, добавить и скобки. Для условных операторов, вложенных в части *THEN* других условных операторов, мы предлагаем использование перевода строки и отступа, как если бы это были обычные операторы `if`:

```
$leapyear =
    $year % 4 == 0
        ? $year % 100 == 0
            ? $year % 400 == 0
                ? 1
                : 0
            : 1
        : 0;
```

Для условных операторов, вложенных в части *ELSE* предшествующих условных операторов, можно сделать аналогичную вещь:

```
$leapyear =
    $year % 4
        ? 0
        : $year % 100
          ? 1
```

```
            : $year % 400
              ? 0
              : 1;
```

но, как правило, лучше построить все части *COND* и *THEN* вертикально:

```
$leapyear =
    $year %   4 ? 0 :
    $year % 100 ? 1 :
    $year % 400 ? 0 : 1;
```

Выравнивание вопросительных знаков и двоеточий может прояснить даже довольно запутанные структуры:

```
printf "Да, мне нравятся мои %s книги!\n",
    $i18n eq "french"   ? "chameau"          :
    $i18n eq "german"   ? "Kamel"            :
    $i18n eq "japanese" ? "\x{99F1}\x{99DD}" :
                          "camel"
```

Можно осуществлять присваивание условному оператору,[1] если второй и третий аргументы являются допустимыми l-значениями (т. е. допускают присваивание им), и оба являются либо скалярами, либо списками (в противном случае Perl не будет знать, какой контекст предоставить правой части присваивания):

```
 ($a_or_b ? $a : $b) = $c;  # либо $a, либо $b получают значение $c
```

Имейте в виду, что условный оператор связывает аргументы сильнее, чем различные операторы присваивания. Обычно это и требуется (взгляните, например, на вышеприведенные присваивания `$leapyear`), но изменить это без помощи скобок нельзя. Присваивание без скобок может привести к неприятностям, при этом можно не получить ошибки синтаксического анализа, поскольку условный оператор может быть проанализирован как левое значение. Например, можно написать так:

```
$a % 2 ? $a += 10 : $a += 2        # НЕВЕРНО
```

Но это будет проанализировано так:

```
(($a % 2) ? ($a += 10) : $a) += 2
```

Операторы присваивания

Perl признает операторы присваивания C, добавляя к ним некоторые собственные. Их много:

---

[1] Не гарантируется, что это улучшит читаемость вашей программы. Но при этом можно создать весомую заявку для победы в конкурсе на самый туманный (obfuscated) код Perl.

```
=    **=    +=    *=    &=    <<=    &&=
            -=    /=    |=    >>=    ||=
            .=    %=    ^=
                  x=
```

Каждый оператор требует l-значения (обычно переменной или элемента массива), являющегося целью, в левой части и выражения в правой части. Для простого оператора присваивания:

```
TARGET = EXPR
```

значение *EXPR* запоминается в переменной или по адресу, указанному в *TARGET*. Для других операторов Perl вычисляет выражение:

```
TARGET OP= EXPR
```

как если бы оно было записано:

```
TARGET = TARGET OP EXPR
```

Это удобное мысленное правило, но в двух отношениях оно вводит в заблуждение. Во-первых, операторы присваивания всегда анализируются на уровне старшинства обычного присваивания, независимо от приоритета, который *OP* имел бы самостоятельно. Во-вторых, *TARGET* вычисляется только один раз. Обычно это не имеет значения, если нет побочных эффектов, таких как автоинкрементирование:

```
$var[$a++] += $value;               # $a инкрементируется один раз
$var[$a++] = $var[$a++] + $value;   # $a инкрементируется два раза
```

В отличие от C, оператор присваивания создает допустимое l-значение. Модификация присваивания аналогична выполнению присваивания с последующей модификацией переменной, которой было присвоено значение. Это удобно при модификации копии чего-либо, например:

```
($tmp = $global) += $constant;
```

что эквивалентно:

```
$tmp = $global + $constant;
```

Аналогично:

```
($a += 2) *= 3;
```

эквивалентно:

```
$a += 2;
$a *= 3;
```

Это не Бог весть как удобно, но вот идиома, которую можно встретить часто:

```
($new = $old) =~ s/foo/bar/g;
```

Во всех случаях значением оператора присваивания является новое значение переменной. Поскольку операторы присваивания ассоциативны справа налево, это можно использовать для присваивания нескольким переменным одного и того же значения:

```
$a = $b = $c = 0;
```

При этом 0 присваивается $c, результат этого действия (по-прежнему 0) присваивается $b, а результат этого действия (*по-прежнему* 0) присваивается $a.

Присваивание списку может выполняться только простым оператором присваивания =. В списковом контексте присваивание списку возвращает список новых значений – так же, как это делает скалярное присваивание. В скалярном контексте присваивание списку возвращает число значений, имевшихся в правой части присваивания, как отмечалось в главе 2 «Всякая всячина». Благодаря этому такое присваивание удобно использовать для проверки значений функций, возвращающих пустой список, когда их вызов оказывается безуспешным (или перестает быть успешным), например:

```
while (($key, $value) = each %gloss) { ... }
next unless ($dev, $ino, $mode) = stat $file;
```

Оператор запятой

Бинарный оператор , является оператором запятой. Находясь в скалярном контексте, он вычисляет свой левый аргумент в пустом контексте, отбрасывает полученное значение, затем вычисляет свой правый аргумент в скалярном контексте и возвращает полученное значение. Это действие вполне идентично действию оператора запятой в C. Например:

```
$a = (1, 3);
```

присваивает $a значение 3. Не следует путать использование этого оператора в скалярном и списковом контекстах. В списковом контексте запятая служит просто разделителем аргументов и вставляет оба свои аргумента в *LIST*. Никакие значения она не отбрасывает.

Например, если изменить предыдущий пример следующим образом:

```
@a = (1, 3);
```

то будет создан список из двух элементов, а

```
atan2(1, 3);
```

является вызовом функции atan2 с двумя аргументами.

Диграф => служит в основном синонимом оператора запятой. Он удобен при записи аргументов, которые задаются парами. Кроме того, он заставляет интерпретировать любой идентификатор, находящийся непосредственно слева от него, как строку.

Списковые операторы (справа)

Правая часть спискового оператора управляет всеми аргументами спискового оператора, которые разделяются запятыми, поэтому приоритет спискового оператора ниже, чем запятой, если смотреть в правую сторону. Когда списковый оператор начинает перемалывать аргументы, разделяемые запятыми, остановить его могут только лексемы, завершающие целое выражение (например, точка с запятой или модификатор команды), либо лексемы, завершающие текущее подвыражение (например, правая круглая или квадратная скобка), либо логические операторы с низким приоритетом, о которых мы и поговорим в следующем разделе.

Логические and, or, not и xor

В качестве альтернативы для `&&`, `||` и `!` Perl предоставляет операторы `and`, `or` и `not`, имеющие низкий приоритет. Способ действия этих операторов идентичен; в частности, `and` и `or` обладают свойством сокращенного действия, как и их аналоги, что делает их полезными не только в логических выражениях, но и для управления логикой программы.

Поскольку приоритет этих операторов значительно ниже, чем у заимствованных из C, их можно безопасно использовать после списковых операторов без всяких скобок:

```
unlink "alpha", "beta", "gamma"
        or gripe(), next LINE;
```

Применяя операторы в стиле C, следовало бы написать нечто типа:

```
unlink("alpha", "beta", "gamma")
        || (gripe(), next LINE);
```

Но нельзя просто взять и заменить все вхождения `||` на `or`. Допустим, что команду

```
$xyz = $x || $y || $z;
```

мы заменим такой:

```
$xyz = $x or $y or $z;    # НЕВЕРНО
```

Она сделает совсем другое! Приоритет присваивания выше, чем приоритет `or`, но ниже, чем приоритет `||`, поэтому `$x` будет присвоена `$xyz`, а затем выполнены `or`. Чтобы получить такой же результат, как с использованием `||`, нужно написать:

```
$xyz = ( $x or $y or $z );
```

Мораль истории такова: выучить приоритеты (или использовать скобки) придется вне зависимости от того, с какими логическими операторами вы работаете.

Существует также логический оператор `xor`, не имеющий точного аналога в C или Perl, поскольку единственный предоставляемый ими оператор исключающего ИЛИ (^) действует поразрядно. Оператор `xor` не может иметь сокращенного действия, поскольку необходимо вычисление операндов с обеих сторон. Лучшим эквивалентом для `$a xor $b` является, вероятно, `!$a != !$b`. Конечно, можно написать `!$a ^ !$b` или даже `$a ? !$b : !!$b`. Суть в том, что `$a` и `$b` должны быть вычислены как true или false в булевом контексте, а имеющийся поразрядный оператор не предоставляет булев контекст без дополнительных усилий.

Операторы C, отсутствующие в Perl

Вот что есть в C и чего нет в Perl:

унарный `&`

Оператор взятия адреса. Однако оператор \ в Perl (для получения ссылки) занимает ту же экологическую нишу:

```
$ref_to_var = \$var;
```

Но ссылки Perl значительно безопаснее, чем указатели C.

унарный `*`

Оператор разыменования адреса. Поскольку в Perl нет адресов, нет нужды и в их разыменовании. Однако ссылки в нем есть, поэтому символы префиксов переменных Perl служат как операторы разыменования, а также указывают на тип: $, @, % и &. Несколько странно, что фактически имеется оператор разыменования *, но поскольку * является специальным символом, обозначающим typeglob, он используется иным образом.

`(TYPE)`

Оператор приведения типа. Во всяком случае, никому не хочется, чтобы его тип приводили.

4

Операторы и объявления

Программа на Perl состоит из последовательности объявлений (declarations) и операторов (statements). Объявление может находиться в любом месте, где допускается команда, но его основное воздействие относится к этапу компиляции. Есть объявления, несущие двойную нагрузку и выполняющие также роль обычных операторов, но большинство из них совершенно прозрачны на этапе исполнения. После компиляции главная последовательность операторов выполняется только один раз.

В отличие от большинства языков программирования, Perl не требует, чтобы переменные объявлялись явным образом. Они начинают существовать при первом обращении к ним, независимо от того, были они объявлены или нет. При попытке использовать значение переменной, которой ни разу не было присвоено значение, она по умолчанию считается нулем (если выступает как число) или `""` – нулевой строкой (если используется как строка), или получает значение «ложь» при обращении к ней как к логической переменной. Если программист хочет получать предупреждения об использовании неопределенных значений как если бы они действительно были строками или числами или даже рассматривать такие действия как ошибку, то об этом позаботится объявление `use warnings` (см. раздел «Директивы» в конце данной главы).

Однако *разрешается* объявить переменную, если есть такое желание, поместив перед ее именем слово `my` или `our`. Программист даже может установить, что использование необъявленной переменной является ошибкой. Это очень хорошо, что соблюдается такой порядок, но необходимо объявить, что этот порядок вам нужен. Обычно Perl не вмешивается в ваши привычки про-

граммирования, но при наличии объявления `use strict` использование необъявленных переменных перехватывается на этапе компиляции. Опять же, см. раздел «Директивы».

Простые операторы

Простой оператор – это выражение, вычисляемое ради его побочных эффектов. Каждый простой оператор должен заканчиваться точкой с запятой, если он не является последним оператором в блоке. В последнем случае точка с запятой не обязательна – Perl понимает, что оператор завершен, поскольку закончился блок. Однако в конце блока из нескольких строк точку с запятой лучше поставить, поскольку потом, возможно, будет добавлена еще одна строка.

Несмотря на то что такие операторы, как `eval {}`, `do {}` и `sub {}`, выглядят как составные операторы, в действительности они таковыми не являются. Верно, внутри них допускается наличие нескольких команд, но это не в счет. При взгляде извне эти операторы являются просто термами в выражении и потому требуют явного использования точки с запятой, если используются как последний элемент оператора.

За любым простым оператором может следовать один необязательный модификатор, как раз перед завершающей точкой с запятой (или окончанием блока). Допускаются следующие модификаторы:

```
if EXPR
unless EXPR
while EXPR
until EXPR
foreach LIST
```

Модификаторы `if` (если) и `unless` (если только не) действуют весьма схожим с обычным языком образом:

```
$trash->take('out') if $you_love_me;
shutup() unless $you_want_me_to_leave;
```

Модификаторы `while` и `until` вычисляются многократно. Модификатор `while` обеспечивает повторное выполнение, пока его выражение остается истинным, а модификатор `until` обеспечивает повторное выполнение, пока его выражение остается ложным:

```
$expression++ while -e "$file$expression";
kiss('me') until $I_die;
```

Модификатор `foreach` (вариант написания `for`) выполняет одно вычисление для каждого элемента в его списке *LIST*, при этом `$_` служит псевдонимом текущего элемента:

```
s/java/perl/ for @resumes;
print "поле: $_\n" foreach split /:/, $dataline;
```

Модификаторы while и until имеют обычную семантику цикла while (сначала вычисляется условие), за исключением применения к do *BLOCK* (или команде do *SUBROUTINE*, использование которой осуждается), когда блок один раз выполняется перед тем, как вычисляется условие. Благодаря этому можно писать такие циклы:

```
do {
    $line = <STDIN>;
    ...
} until $line eq ".\n";
```

Посмотрите три разных описания do в главе 29 «Функции». Обратите также внимание на то, что операторы управления циклом, описываемые ниже, в этой конструкции не будут работать, поскольку модификаторы не принимают метки циклов. Всегда можно поместить вокруг нее лишний блок для досрочного завершения или внутри нее для досрочной итерации, как описывается далее в разделе «Голые блоки». Либо можно написать действительный цикл с несколькими операторами управления циклом внутри. Если речь зашла о настоящих циклах, то нужно сначала поговорить о составных командах.

Составные операторы

Последовательность операторов в области видимости[1] называется *блоком*. Иногда областью видимости является весь файл, например файл, загружаемый директивой require, или файл, содержащий главную программу. Иногда областью видимости является строка, вычисляемая через eval. Но обычно блок – это то, что заключено в фигурные скобки ({}). Когда говорят об области видимости, то имеют в виду любой из этих трех вариантов. Когда имеют в виду блок в фигурных скобках, то используют термин «блок» (*BLOCK*).

Составные операторы образуются из выражений и блоков (*BLOCK*). Выражения состоят из термов и операторов. В наших синтаксических описаниях мы будем словом *EXPR* обозначать место, в котором можно использовать любое скалярное выражение. Для обозначения выражения, вычисляемого в списковом контексте, будем говорить *LIST*.

Следующие команды можно применять для управления условным и многократным выполнением блоков *BLOCK*. (Часть *LABEL* является необязательной.)

```
if (EXPR) BLOCK
if (EXPR) BLOCK else BLOCK
if (EXPR) BLOCK elsif (EXPR) BLOCK ...
if (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK
```

[1] Области видимости и пространства имен описываются в разделе «Имена» главы 2 «Всякая всячина».

```
unless (EXPR) BLOCK
unless (EXPR) BLOCK else BLOCK
unless (EXPR) BLOCK elsif (EXPR) BLOCK ...
unless (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK

LABEL while (EXPR) BLOCK
LABEL while (EXPR) BLOCK continue BLOCK

LABEL until (EXPR) BLOCK
LABEL until (EXPR) BLOCK continue BLOCK

LABEL for (EXPR; EXPR; EXPR) BLOCK

LABEL foreach (LIST) BLOCK
LABEL foreach VAR (LIST) BLOCK
LABEL foreach VAR (LIST) BLOCK continue BLOCK

LABEL BLOCK
LABEL BLOCK continue BLOCK
```

Обратите внимание на то, что, в отличие от C и Java, они определены в терминах блоков, а не операторов. Это означает, что фигурные скобки необходимы – наличие висящих операторов не допускается. Условный оператор без фигурных скобок можно записать несколькими способами. Все следующие команды делают одно и то же:

```
unless (open(FOO, $foo))    { die "Невозможно открыть $foo: $!" }
if (!open(FOO, $foo))       { die "Невозможно открыть $foo: $!" }

die "Невозможно открыть $foo: $!"   unless open(FOO, $foo);
die "Невозможно открыть $foo: $!"   if !open(FOO, $foo);

open(FOO, $foo)             || die "Невозможно открыть $foo: $!";
open FOO, $foo              or die "Невозможно открыть $foo: $!";
```

В большинстве случаев мы предпочитаем последнюю пару форматов. Они выглядят менее перегруженными, чем остальные, особенно вариант `or die`. В формате `||` нужно привыкнуть к скрупулезному указанию кавычек, но в версии `or` о них можно и позабыть.

Но главная причина, по которой нам больше нравятся последние варианты, состоит в том, что они выдвигают в начало строки самую важную часть оператора, где ее сразу видно. Обработка ошибки смещается в сторону, так что можно не обращать на нее внимания, пока в этом не возникнет необходимость.[1] Если каждый раз табулировать все проверки `or die`, чтобы они оказывались в одной и той же колонке справа, то чтение окажется еще более легким:

```
chdir $dir                  or die "Смена каталога $dir: $!";
open FOO, $file             or die "Открытие $file: $!";
```

[1] (Как в этой сноске.)

```
@lines = <FOO>              or die "$file пуст?";
close FOO                   or die "Закрытие $file: $!";
```

Операторы if и unless

Оператор `if` прост. Поскольку блоки всегда заключены в фигурные скобки, неоднозначности в отношении того, к которому `if` относится тот или иной `else` или `elsif`, не возникает. Из каждой данной последовательности блоков `if/elsif/else` выполняется только тот первый, значение условия которого истинно. Если ни одно условие не является истинным, то выполняется блок `else`, если он имеется. Обычно полезно поместить `else` в конце цепочки `elsif` на тот случай, если один из вариантов будет случайно пропущен.

Если `unless` используется вместо `if`, то смысл проверки условия становится противоположным. Это значит, что:

```
unless ($x == 1) ...
```

равносильно:

```
if ($x != 1) ...
```

или даже неприглядному:

```
if (!($x == 1)) ...
```

Область видимости переменной, объявленной в управляющем условии, простирается от места ее объявления и по всему условному оператору, включая все `elsif` и завершающее предложение `else`, если они имеются, но не далее того:

```
if ((my $color = <STDIN>) =~ /red/i) {
    $value = 0xff0000;
}

elsif ($color =~ /green/i) {
    $value = 0x00ff00;
}

elsif ($color =~ /blue/i) {
    $value = 0x0000ff;
}

else {
    warn "неизвестная составляющая RGB `$color', вместо нее использован черный \n";
    $value = 0x000000;
}
```

По завершении блока `else` переменная `$color` больше не видима. Если нужно продлить область видимости, объявите переменную заранее.

Операторы Loop

Формальный синтаксис всех операторов цикла допускает использование необязательной метки *LABEL*. (Метку можно поставить против любого оператора, но в цикле она имеет особое значение.) Метка состоит из идентификатора и двоеточия, следующего за ним. Обычно метки пишутся в верхнем регистре, чтобы избежать конфликта с зарезервированными словами и для лучшего выделения. Perl не будет сбит с толку, если использовать метку, уже имеющую значение, например метку `if` или `open`, чего нельзя сказать о тех, кто будет читать вашу программу.

Операторы while и until

Оператор `while` повторно выполняет блок, пока *EXPR* остается истинным. Если слово `while` заменить словом `until`, то смысл проверки становится противоположным, т. е. блок выполняется, только пока *EXPR* имеет значение «ложь». Однако условие все равно проверяется перед первой итерацией цикла.

Операторы `while` или `until` могут иметь необязательный дополнительный блок: блок `continue`, выполняемый тогда, когда блок продолжается – при выходе за конец первого блока или явном вызове `next` (оператор управления циклом, который вызывает переход к новой итерации). На практике блок `continue` применяется не слишком часто, но, сказав о нем, мы теперь сможем строго определить цикл `for` в следующем разделе.

В отличие от цикла `foreach`, который мы рассмотрим несколько ниже, цикл `while` не локализует переменные в проверяемом условии неявным образом. В результате могут возникнуть «интересные» последствия, если в качестве переменных цикла используются глобальные переменные. В частности, посмотрите в разделе «Оператор ввода строки (угловые скобки)» главы 2 «Всякая всячина», как в некоторых циклах `while` может происходить неявное присвоение глобальной переменной `$_`, а также пример того, как решить эту проблему, явно делая `$_` локальной. Другие переменные цикла лучше всего объявлять с помощью ключевого слова `my`, как показано в следующем примере.

Переменная, объявленная в условии оператора `while` или `until`, имеет видимость только в одном или нескольких блоках, на которые распространяется действие этого условия. Она не входит в окружающую область видимости. Например:

```
while (my $line = <STDIN>) {
    $line = lc $line;
}
continue {
    print $line;   # пока видима
}
# теперь $line ушла из области видимости
```

В данном случае область видимости `$line` простирается от ее объявления в управляющем выражении через всю конструкцию цикла, включая блок

continue, но не далее того. Если необходимо продлить область видимости, объявите переменную перед циклом.

Циклы for

Состоящий из трех частей цикл for содержит в скобках выражения, разделенные точкой с запятой. Эти выражения выступают, соответственно, как выражения инициализации, условия и реинициализации цикла. Все три выражения (но не точки с запятыми) не обязательны; условие, если оно опущено, всегда считается истинным. Таким образом, состоящий из трех частей цикл for можно определить на языке соответствующего цикла while. Оператор:

```
LABEL:
  for (my $i = 1; $i <= 10; $i++) {
      ...
  }
```

аналогичен следующему:

```
{
    my $i = 1;
  LABEL:
    while ($i <= 10) {
        ...
    }
    continue {
        $i++;
    }
}
```

за исключением того, что внешнего блока на самом деле нет, и мы обозначили его только для того, чтобы показать границы области видимости my.

Если необходимо производить итерацию одновременно по двум переменным, просто разделите параллельные выражения запятыми:

```
for ($i = 0, $bit = 0; $i < 32; $i++, $bit <<= 1) {
    print "Бит $i установлен \n" if $mask & $bit;
}
# значения $i и $bit сохраняются после конца цикла
```

Можно также объявить эти переменные так, чтобы они оказались видимыми только внутри цикла for:

```
for (my ($i, $bit) = (0, 1); $i < 32; $i++, $bit <<= 1) {
    print "Бит $i установлен \n" if $mask & $bit;
}
# теперь версии $i и $bit для цикла вышли из области видимости
```

Кроме обычного перебора индексов массива, цикл for может быть использован для многих других интересных приложений. Ему даже не требуется яв-

ная переменная цикла. Вот пример того, как можно избежать проблемы, возникающей при явной проверке конца файла в интерактивном дескрипторе файла и приводящей программу к зависанию.

```
$on_a_tty = -t STDIN && -t STDOUT;
sub prompt { print "да? " if $on_a_tty }
for ( prompt(); <STDIN>; prompt() ) {
    # какие-то действия
}
```

Еще одно традиционное применение цикла `for`, состоящего из трех частей, связано с тем, что все три выражения не обязательны, а условие по умолчанию является истинным. Если опустить все три выражения, то получится бесконечный цикл:

```
for (;;) {
    ...
}
```

Это то же самое, что

```
while (1) {
    ...
}
```

Если вас смущает идея бесконечного цикла, то следует заметить, что всегда можно выйти из цикла, находясь в любом его месте, с помощью явного оператора управления циклом, такого как `last`. Конечно, если разрабатывается управляющая программа для крылатой ракеты, то явный выход из цикла может не потребоваться. В должное время цикл будет завершен автоматически.[1]

Циклы foreach

Цикл `foreach` осуществляет перебор списка значений, последовательно устанавливая управляющую переменную (*VAR*) равной каждому элементу списка:

```
foreach VAR (LIST) {
    ...
}
```

Ключевое слово `foreach` служит просто синонимом ключевого слова `for`, поэтому `for` и `foreach` можно использовать взаимозаменяемо и в зависимости от того, которое из них выглядит более удобочитаемым в конкретной ситуации. Если *VAR* опущена, используется глобальная переменная `$_`. (Не беспокойтесь, Perl легко отличит `for (@ARGV)` от `for ($i=0; $i<$#ARGV; $i++)`, поскольку в последнем содержатся точки с запятой.) Вот несколько примеров:

[1] То есть имеется тенденция к автоматическому выходу из цикла.

```
$sum = 0; foreach $value (@array) { $sum += $value }

for $count (10,9,8,7,6,5,4,3,2,1,'БУМ!') {  # обратный счет
    print "$count\n"; sleep(1);
}

for (reverse 'БУМ!', 1 .. 10) {             # то же самое
    print "$_\n"; sleep(1);
}

for $field (split /:/, $data) {             # любое выражение LIST
    print "Поле содержит: `$field'\n";
}

foreach $key (sort keys %hash) {
    print "$key => $hash{$key}\n";
}
```

Последний пример представляет собой канонический способ вывода значений хеша в отсортированном порядке. Более подробные примеры можно найти в описаниях `keys` и `sort` в главе 29 «Функции».

Применяя `foreach`, невозможно узнать, в каком месте списка вы находитесь. Можно сравнивать соседние элементы списка, запоминая предыдущий элемент в какой-нибудь переменной, но иногда нужно просто взять и написать трехчастный цикл `for`, перебирающий индексы. В конце концов, для того и создан цикл `for` второго типа.

Если *LIST* состоит только из значений, допускающих присвоение (т. е. обычно переменных, а не перечисления констант), все эти переменные могут быть модифицированы путем изменения *VAR* внутри цикла. Это возможно потому, что индексная переменная цикла `foreach` представляет собой неявный псевдоним для каждого элемента обрабатываемого списка. Можно изменить не только один массив, а сразу несколько массивов и хешей, находящихся в одном списке:

```
foreach $pay (@salaries) {                  # всем прибавка 8%
    $pay *= 1.08;
}

for (@christmas, @easter) {                 # изменить меню
    s/ham/turkey/;
}

s/ham/turkey/ for @christmas, @easter;      # то же
for ($scalar, @array, values %hash) {
    s/^\s+//;                               # удалить ведущие пробелы
    s/\s+$//;                               # удалить замыкающие пробелы
}
```

Переменная цикла действует только в динамической или лексической области видимости цикла и неявно становится лексической, если предвари-

тельно объявлена с помощью my. Благодаря этому она невидима для всех функций, определенных вне области лексической видимости этой переменной, даже если они вызываются из этого цикла. Однако если в области видимости нет лексического объявления, то переменной цикла будет локализованная (с динамической областью видимости) глобальная переменная; это позволяет функциям, вызываемым из цикла, обращаться к этой переменной. В любом случае предыдущее значение, которое локализованная переменная имела до цикла, будет восстановлено при выходе из него.

При желании можно явно определить тип (лексический или глобальный) используемой переменной. Это облегчит работу тех, кто будет сопровождать ваш код; в противном случае им придется искать в охватывающих областях видимости предыдущее объявление, чтобы определить тип этой переменной:

```
for my  $i    (1 .. 10) { ... }         # $i всегда лексическая
for our $Tick (1 .. 10) { ... }         # $Tick всегда глобальная
```

Если переменная цикла сопровождается объявлением, то предпочтительнее использовать более краткую форму for, а не foreach, поскольку она лучше читается на английском языке.

Вот как программист C или Java мог бы поначалу решить запрограммировать некоторый алгоритм на Perl:

```
for ($i = 0; $i < @ary1; $i++) {
    for ($j = 0; $j < @ary2; $j++) {
        if ($ary1[$i] > $ary2[$j]) {
            last;         # Нельзя выйти во внешний цикл. :-(
        }
        $ary1[$i] += $ary2[$j];
    }
    # сюда приведет этот last
}
```

А вот как поступил бы ветеран программирования на Perl:

```
WID: foreach $this (@ary1) {
    JET: foreach $that (@ary2) {
        next WID if $this > $that;
        $this += $that;
    }
}
```

Видите, насколько легче это делается с помощью идиом Perl? Аккуратнее, надежнее и быстрее. Аккуратнее, поскольку меньше шума. Надежнее, поскольку если позже добавить код между внутренним и внешним циклами, новый код не будет выполнен по ошибке, так как next (о котором говорится ниже) явно проходит внешний цикл, а не просто выходит из внутреннего. И быстрее, поскольку Perl выполняет оператор foreach быстрее, чем эквивалентный ему цикл for, т. к. обращение к элементам выполняется непосредственно, а не через индексы.

Однако пишите так, как вам больше нравится. TMTOWTDI.

Как и оператор `while`, оператор `foreach` может иметь блок `continue`. Это позволяет выполнять фрагмент кода в конце каждой итерации цикла вне зависимости от того, попадаете вы в него при обычном ходе событий или посредством `next`.

И теперь мы можем, наконец, сказать: `next` будет следующим.

Управление циклом

Мы уже говорили о том, что можно поместить метку на цикле и тем самым дать ему имя. Метка цикла идентифицирует его для операторов управления циклом, таких как `next`, `last` и `redo`. Метка дает имя циклу в целом, не только его вершине. Поэтому оператор управления циклом, ссылаясь на цикл, на самом деле не осуществляет переход «go to» на метку цикла. С точки зрения компьютера, с таким же успехом метку можно было поместить и в конце цикла, но по каким-то причинам люди предпочитают, чтобы метка стояла вверху.

Обычно циклам дают имя по тому элементу, который цикл обрабатывает при каждой итерации. Это прекрасно уживается с операторами управления циклом, которые задуманы так, чтобы их можно было читать как обычный английский язык при использовании надлежащих меток и модификаторов операторов. Исконный цикл работает со строками, поэтому исконной меткой цикла является `LINE:`, а исконным оператором управления циклом нечто типа:

```
next LINE if /^#/;      # выкинуть комментарии
```

Синтаксис операторов управления циклом следующий:

```
last LABEL
next LABEL
redo LABEL
```

Метка *LABEL* является необязательной; если она опущена, то оператор ссылается на самый внутренний из охватывающих циклов. Но если нужно перескочить более чем через один уровень, вы должны использовать метку, чтобы указать цикл, на который хотите оказать воздействие. Эта метка не обязана попадать в вашу лексическую область видимости, хотя, вероятно, ей следовало бы там находиться. На самом деле метка может находиться в любом месте вашей динамической области видимости. Если при этом происходит выход из `eval` или подпрограммы, Perl выдаст предупредительное сообщение (если его об этом попросить).

Так же как можно иметь в функции сколько угодно операторов `return`, в цикле можно иметь сколько угодно операторов управления циклом. Не следует считать это плохим тоном. На заре структурного программирования некоторые настаивали на том, чтобы в циклах и программах имелось только по одному входу и одному выходу. Концепция одного входа по-прежнему

остается правильной идеей, однако концепция одного выхода приводит к тому, что приходится писать много неестественного кода. Задачей программирования часто является обход дерева решений. Дерево решений обычно начинается с одного ствола и заканчивается многочисленными листьями. Вставляйте в код столько выходов из циклов (и возвратов из функций), сколько свойственно решаемой вами задаче. Если области видимости переменных определены разумно, то в нужный момент все автоматически очищается независимо от того, каким образом выполняется выход из блока.

Оператор `last` осуществляет немедленный выход из соответствующего цикла. Блок `continue`, если он есть, не выполняется. В следующем примере осуществляется «катапультирование» из цикла после обнаружения первой пустой строки:

```
LINE: while (<STDIN>) {
    last LINE if /^$/;      # выйти после обработки почтового заголовка
    ...
}
```

Оператор `next` пропускает оставшуюся часть текущей итерации (прохода) цикла и начинает новую итерацию. Если в цикле есть предложение `continue`, то оно выполняется перед тем, как заново вычислить условие, точно так же, как третья составляющая трехчастного цикла `for`. Поэтому `continue` можно применять для инкрементирования переменной цикла, даже если данная итерация цикла была прервана из-за `next`:

```
LINE: while (<STDIN>) {
    next LINE if /^#/;      # пропустить комментарии
    next LINE if /^$/;      # пропустить пустые строки
    ...
} continue {
    $count++;
}
```

Оператор `redo` перезапускает блок цикла, не проверяя снова условия. Блок `continue`, если он есть, не выполняется. Этот оператор часто используется в программах, которым требуется «обмануть» себя, читая данные. Предположим, что обрабатывается файл, в котором строка может завершаться обратной косой чертой, означающей продолжение записи на следующей строчке. Вот как в этом случае можно использовать `redo`:

```
while (<>) {
    chomp;
    if (s/\\$//) {
        $_ .= <>;
        redo unless eof;    # не читать после eof каждого файла

    }
    # теперь обработать $_

}
```

Это обычное сокращение Perl для более явного (и скучного) варианта:

```
LINE: while (defined($line = <ARGV>)) {
    chomp($line);
    if ($line =~ s/\\$//) {
        $line .= <ARGV>;
        redo LINE unless eof(ARGV);
    }
    # теперь обработать $line
}
```

Вот пример из реальной программы, использующей все три оператора управления циклом. Хотя данная стратегия анализа аргументов командной строки реже применяется сейчас, когда с Perl поставляются модули `Getopts::*`, она все же представляет собой прекрасную иллюстрацию использования операторов управления циклом для вложенных циклов с именами:

```
ARG: while (@ARGV && $ARGV[0] =~ s/^-(?=.)//) {
    OPT: for (shift @ARGV) {
         m/^$/       && do {                            next ARG; };
         m/^-$/      && do {                            last ARG; };
         s/^d//      && do { $Debug_Level++;            redo OPT; };
         s/^l//      && do { $Generate_Listing++;       redo OPT; };
         s/^i(.*)//  && do { $In_Place = $1 || ".bak";  next ARG; };
         say_usage("Неизвестная опция: $_");
    }
}
```

Еще одно замечание относительно операторов управления циклом. Как можно было заметить, мы не называем их утверждениями (statements). Это связано с тем, что они не являются утверждениями, хотя, как всякое выражение, могут быть использованы как утверждение. Можно даже представить их себе как унарные операторы, которым дозволено изменять порядок выполнения программы. Поэтому их допускается применять везде, где имеет смысл их использование в выражении. На самом деле операторы управления циклом можно использовать даже там, где это не имеет смысла. Иногда можно видеть в программе такую ошибку:

```
open FILE, $file
     or warn "Невозможно открыть $file: $!\n", next FILE;   # НЕВЕРНО
```

Намерения благие, но `next FILE` будет проанализирован как аргумент `warn`, являющегося списковым оператором. Поэтому `next` выполняется раньше, чем `warn` получает возможность выдать предупреждение. В данном случае ситуацию легко исправить, обратив списковый оператор `warn` в вызов функции `warn` с помощью надлежащим образом расставленных скобок:

```
open FILE, $file
     or warn("Невозможно открыть $file: $!\n"), next FILE;   # okay
```

Однако может показаться, что следующий код прочесть легче:

```
unless (open FILE, $file) {
     warn "Невозможно открыть $file: $!\n";
     next FILE;
}
```

Голые блоки

Отдельный блок, независимо от наличия у него метки, семантически эквивалентен однократно выполняемому циклу. Поэтому можно использовать `last` для выхода из блока или `redo` для его повторного выполнения.[1] Заметьте, что это не так для блоков в `eval {}`, `sub {}` и, ко всеобщему удивлению, `do {}`. Эти три конструкции – не блоки циклов, поскольку сами не являются блоками (*BLOCK*s); стоящее впереди ключевое слово делает их просто термами в выражении, в которых оказался блок кода. Поскольку они не являются блоками циклов, им нельзя присвоить метку или применить операторы управления циклом. Операторы управления циклом можно использовать только в настоящих циклах, так же как `return` – только в подпрограмме (ну, еще в `eval`).

Операторы управления циклом не работают также в `if` и `unless`, поскольку это не циклы. Но всегда можно вставить дополнительную пару фигурных скобок, чтобы создать голый блок, который *считается* циклом:

```
if (/pattern/) {{
    last if /alpha/;
    last if /beta/;
    last if /gamma/;
    # какие-то действия, если все еще в if()
}}
```

Вот каким образом блок обеспечивает операторам управления циклом возможность работать с конструкцией `do {}`. Для того чтобы использовать `next` или `redo` в `do`, поместите внутрь голый блок:

```
do {{
    next if $x == $y;
    # здесь какие-то действия
}} until $x++ > $z;
```

Для применения `last` нужно потрудиться больше:

```
{
    do {
        last if $x = $y ** 2;
```

[1] По причинам, которые, возможно, прояснятся (а возможно, и нет) после некоторых размышлений, оператор `next` тоже осуществляет выход из однократно выполняемого блока. Однако есть некоторое различие: `next` выполнит блок `continue`, а `last` – нет.

```
        # здесь какие-то действия
    } while $x++ <= $z;
}
```

А если требуется обеспечить возможность работы с обоими операторами управления циклом, то придется поставить метки на эти блоки, чтобы их можно было отличить друг от друга:

```
DO_LAST: {
            do {
DO_NEXT:          {
                    next DO_NEXT if $x == $y;
                    last DO_LAST if $x =  $y ** 2;
                    # здесь какие-то действия
                  }
            } while $x++ <= $z;
         }
```

Но, дойдя до такой конструкции (если не раньше), лучше всего организовать обычный бесконечный цикл с `last` в конце:

```
for (;;) {
    next if $x == $y;
    last if $x =  $y ** 2;
    # здесь какие-то действия
    last unless $x++ <= $z;
}
```

Структуры типа case

В отличие от некоторых других языков программирования, в Perl нет официальных операторов `switch` или `case`. Это связано с тем, что в Perl они и не требуются, т. к. он предоставляет много способов делать то же самое, что и эти операторы. Особенно удобны для создания структур типа case (многовариантных переключателей) голые блоки. Вот один пример:

```
SWITCH: {
    if (/^abc/) { $abc = 1; last SWITCH; }
    if (/^def/) { $def = 1; last SWITCH; }
    if (/^xyz/) { $xyz = 1; last SWITCH; }
    $nothing = 1;
}
```

а вот другой:

```
SWITCH: {
    /^abc/      && do { $abc = 1; last SWITCH; };
    /^def/      && do { $def = 1; last SWITCH; };
    /^xyz/      && do { $xyz = 1; last SWITCH; };
    $nothing = 1;
}
```

или с таким форматированием, которое лучше выделяет каждый вариант:

```
SWITCH: {
    /^abc/      && do {
                        $abc = 1;
                        last SWITCH;
                    };
    /^def/      && do {
                        $def = 1;
                        last SWITCH;
                    };
    /^xyz/      && do {
                        $xyz = 1;
                        last SWITCH;
                     };
    $nothing = 1;
}
```

или даже (о, ужас!):

```
if    (/^abc/) { $abc = 1 }
elsif (/^def/) { $def = 1 }
elsif (/^xyz/) { $xyz = 1 }
else           { $nothing = 1 }
```

В следующем примере обратите внимание на то, как операторы `last` игнорируют блоки `do {}`, которые не являются циклами, и осуществляют выход из цикла `for`:

```
for ($very_nasty_long_name[$i++][$j++]->method()) {
    /this pattern/      and do { push @flags, '-e'; last; };
    /that one/    and do { push @flags, '-h'; last; };
    /something else/       and do {                    last; };
    die "неизвестное значение: `$_'";
}
```

Вам может показаться странным, что организуется цикл по единственному значению, поскольку проход через цикл будет совершен только один раз. Однако удобно использовать способность `for/foreach` создавать псевдонимы для того, чтобы осуществить временное, локальное присвоение для `$_`. При повторяющихся сравнениях с одним и тем же длинным значением это весьма облегчает ввод с клавиатуры и, соответственно, затрудняет возникновение опечаток. Устраняется возможность получения побочного эффекта от повторного вычисления выражения. А что касается данного раздела, то это одна из наиболее часто встречающихся стандартных идиом, реализующих структуру `switch` или `case`.

В простых структурах выбора можно также использовать оператор `?:` каскадным образом. В следующем примере мы снова прибегаем к `for` и его свойству создания псевдонимов, чтобы улучшить читаемость повторных сравнений:

```
for ($user_color_preference) {
    $value = /red/      ?   0xFF0000  :
             /green/    ?   0x00FF00  :
             /blue/     ?   0x0000FF  :
                            0x000000  ;   # черный, если нет совпадений
}
```

В таких случаях, как последний, иногда лучше создать свой хеш и быстро найти в нем ответ по индексу. В отличие от только что показанного каскадирования условных операторов, хеш может содержать неограниченное число элементов, и время поиска последнего из них ничуть не больше, чем первого. Недостаток в том, что можно использовать только точный поиск, но не соответствие шаблону. Если есть такой хеш:

```
%color_map = (
    azure       => 0xF0FFFF,
    chartreuse  => 0x7FFF00,
    lavender    => 0xE6E6FA,
    magenta     => 0xFF00FF,
    turquoise   => 0x40E0D0,
);
```

то точный поиск строки выполняется быстро:

```
$value = $color_map{ lc $user_color_preference } || 0x000000;
```

Даже сложные многократно ветвящиеся операторы (где в каждом случае требуется выполнить несколько различных операторов) можно превратить в быстрый поиск. Нужно только использовать хеш ссылок на функции. О том, как это делается, рассказано в разделе «Хеши функций» главы 9 «Структуры данных».

goto

Не по малодушию (но и не с легким сердцем) Perl все же поддерживает оператор `goto`. Есть три его вида: `goto` *`LABEL`*, `goto` *`EXPR`* и `goto &`*`NAME`*.

В формате `goto` *`LABEL`* отыскивается оператор с меткой *`LABEL`* и выполнение продолжается с него. Его нельзя применять для перехода в какую-либо конструкцию, требующую инициализации, например, подпрограмму или цикл `foreach`. Нельзя также использовать его для перехода в конструкцию, удаленную в результате оптимизации (см. главу 18 «Компиляция»). С его помощью можно перейти почти в любое другое место в текущем блоке или любом блоке динамической области видимости (т. е. блоке, из которого был произведен вызов). Посредством `goto` можно даже выходить из подпрограмм, но обычно лучше делать это другими средствами. Автор Perl никогда не испытывал нужды использовать эту форму `goto` (имеется в виду, в Perl, в C это другое дело).

Форма goto *EXPR* является просто обобщением goto *LABEL*. Предполагается, что результат вычисления выражения *EXPR* представляет собой имя метки, местонахождение которой, очевидно, должно динамически определяться интерпретатором. В результате можно применять вычисляемые переходы goto, как в FORTRAN, что не особенно рекомендуется делать, если требуется облегчить сопровождение кода:

```
goto(("FOO", "BAR", "GLARCH")[$i]);          # в надежде, что 0 <= i < 3

@loop_label = qw/FOO BAR GLARCH/;
goto $loop_label[rand @loop_label];          # случайная телепортация
```

В большинстве подобных случаев обычно гораздо более правильным решением является применение структурных механизмов управления последовательностью выполнения, предоставляемых next, last и redo, а не использование goto. В некоторых приложениях разумный подход состоит в использовании хеша ссылок на функции или обработки исключительных ситуаций с помощью пары eval и die.

Форма goto &*NAME* является прямо волшебной и достаточно далека от обычного goto, что должно избавить ее пользователей от тех оскорблений, которым обычно подвергаются те, кто не пренебрегает goto. Она заменяет вызов указанной подпрограммы на подпрограмму, выполняющуюся в данный момент. Такое свойство этого оператора используется подпрограммами AUTOLOAD, чтобы загрузить другую подпрограмму, а затем представить дело так, будто в первую очередь была вызвана та, другая подпрограмма. После goto даже функция caller не сможет определить, что первой была вызвана данная подпрограмма. Все модули autouse, AutoLoader и SelfLoader используют эту стратегию, чтобы определять функции при первом к ним обращении, а затем передавать в них управление, так что никто никогда и не заметит, что этих функций в начале не было.

Глобальные объявления

Объявления подпрограмм и форматов являются глобальными. Независимо от того, куда они помещены, объявляемые ими элементы являются глобальными (локальными для пакета, но пакеты глобальны для программы, поэтому все, что находится в пакете, видимо в любом месте). Глобальное объявление можно поместить в любое место, где может находиться оператор, но оно не оказывает влияния на выполнение основной последовательности операторов: действие объявлений сказывается только на этапе компиляции.

Это значит, что нельзя создавать условные объявления подпрограмм или форматов, скрывая их от компилятора в условных операторах этапа исполнения типа if, т. к. только интерпретатор обращает внимание на условное выполнение этих операторов. Объявления подпрограмм и форматов (а также объявления use и no) видны компилятору независимо от того, где они находятся.

Глобальные объявления обычно помещаются в начале или конце программы либо отдельно в другом файле. Однако при объявлении переменных с лекси-

ческой областью видимости (см. следующий раздел) нужно обеспечить, чтобы определение формата или подпрограммы попало в область видимости объявлений тех переменных, доступ к которым вы хотите иметь в этом определении.

Заметьте, что мы незаметно перешли в разговоре с объявлений на определения. Иногда полезно разделить *определение* (*definition*) подпрограммы и ее *объявление* (*declaration*). Единственная синтаксическая разница между ними в том, что определение включает блок, содержащий код, который должен быть выполнен, а в объявлении его нет. (Определение подпрограммы действует как ее объявление, если объявления нет в области видимости.) Отделение определения от объявления позволяет поместить объявление подпрограммы в начало файла, а определение – в конец (при этом ваши объявления переменных с лексической областью видимости удачно располагаются посередине):

```
sub count (@);        # Теперь компилятор знает, как вызывать count().
my $x;                # Теперь компилятор знает о лексической переменной.
$x = count(3,2,1);    # Компилятор может проверить вызов функции.
sub count (@) { @_ } # Теперь компилятор знает, что означает count().
```

Как показывает этот пример, определения подпрограмм фактически не обязательно должны присутствовать, чтобы вызовы их могли быть скомпилированы (в действительности, определение можно даже отложить до первого обращения, если применяется автозагрузка), однако объявление подпрограмм оказывает компилятору различную помощь и предоставляет вам больший выбор в способах их вызова.

Объявление подпрограммы позволяет использовать ее при компиляции, начиная с данного места, без скобок, как если бы это был встроенный оператор. (В последнем примере мы использовали скобки для вызова `count`, но фактически этого не требовалось.) Объявить подпрограмму, не определяя ее, можно, просто сказав:

```
sub myname;
$me = myname $0             or die "Не могу найти myname";
```

Такое голое объявление делает функцию списковым, а не унарным оператором, поэтому будьте внимательны и применяйте `or`, а не `||`. Оператор `||` слишком сильно связывает аргументы, чтобы быть использованным после списковых операторов, хотя всегда можно заключить аргументы спискового оператора в круглые скобки, превратив его снова в нечто, более похожее на вызов функции. Альтернативным вариантом является использование прототипа `($)`, позволяющее превратить подпрограмму в унарный оператор:

```
sub myname ($);
$me = myname $0             || die " Не могу найти myname";
```

Теперь это будет проанализировано так, как вы рассчитываете, но все же следует выработать привычку использовать в такой ситуации `or`. Подробнее о прототипах см. главу 6 «Подпрограммы».

Вы *обязаны* определить подпрограмму в каком-то месте, иначе на этапе исполнения получите ошибку, указывающую на вызов процедуры, которая не определена. Помимо самостоятельного определения подпрограммы есть несколько способов получить определения из других мест.

Можно загружать определения из других файлов с помощью простого оператора `require`. В Perl 4 это был лучший способ загрузки файлов, но с ним связаны две проблемы. Во-первых, другой файл обычно помещает имена подпрограмм в пакет (таблицу имен) по собственному выбору, а не в ваши пакеты. Во-вторых, обращение к `require` имеет место на этапе исполнения, то есть слишком поздно, чтобы служить объявлением в файле, вызывающем `require`. Иногда, впрочем, отложенная загрузка является как раз тем, что требуется.

Более удобный способ загрузки объявлений и определений предоставляет директива `use`, которая, по сути, вызывает `require` для модуля на этапе компиляции (поскольку `use` рассматривается как блок `BEGIN`) и позволяет импортировать некоторые из объявлений модуля в вашу программу. Поэтому можно рассматривать `use` как своего рода глобальное объявление, поскольку она на этапе компиляции импортирует имена в ваш собственный (глобальный) пакет так, как если бы вы объявили их сами. В разделе «Таблицы символов» главы 10 «Пакеты» приведено описание механизма низкого уровня, действующего при импорте из одного пакета в другой; в главе 11 «Модули» – описание настройки импорта и экспорта в модуле; в главе 18 «Компиляция» – описание `BEGIN` и родственных ему `CHECK`, `INIT` и `END`, которые тоже в некотором роде служат глобальными объявлениями, поскольку работа с ними происходит на этапе компиляции и может иметь глобальные результаты.

Объявления с областью видимости

Как и глобальные объявления, объявления с лексической областью видимости оказывают воздействие на этапе компиляции. В отличие от глобальных объявлений, объявления с лексической областью видимости действуют только от точки, где находится объявление, и до конца самой внутренней из охватывающих областей видимости (блока, файла или `eval` – что раньше встретится). Поэтому мы говорим: «с лексической областью видимости», хотя «с текстуальной областью видимости» было бы, вероятно, точнее, поскольку «лексическая область видимости» имеет мало общего со словарями (лексиконами). Однако компьютерные специалисты во всем мире знают, что означает «лексическая область видимости» («lexically scoped»), поэтому мы сохраняем здесь это название.

Perl поддерживает также объявления с динамической областью видимости. *Динамическая область видимости* (*dynamic scope*) тоже распространяется до конца самого внутреннего из охватывающих блоков, но в этом случае «охватывание» динамически определяется на этапе исполнения, а не текстуально на этапе компиляции. Иными словами, динамическая вложенность блоков определяется вызовом одного блока другим, а не включением одного в состав другого. Эта вложенность динамических областей может отчасти кор-

релировать с вложенностью лексических областей, но, вообще говоря, они не идентичны, особенно если вызываются какие-либо подпрограммы.

Мы уже говорили о том, что в некоторых отношениях `use` можно рассматривать в качестве глобального объявления, но в других отношениях `use` имеет лексическую область видимости. В частности, `use` не только импортирует имена пакета, но также осуществляет различные магические подсказки компилятору, называемые *директивами*. Прагмы в большинстве своем имеют лексическую область видимости, включая директиву `use strict 'vars'`, которая требует, чтобы переменные были объявлены перед использованием. См. далее раздел «Прагмы».

Объявление пакета, как ни странно, само имеет лексическую область видимости, несмотря на то, что пакет представляет собой глобальный объект. Но объявление пакета просто объявляет имя пакета по умолчанию для оставшейся части охватывающего блока. В этом пакете отыскиваются необъявленные, неквалифицированные имена переменных.[1] В некотором смысле пакет вовсе и не объявляется, но начинает существовать, когда вы ссылаетесь на нечто, принадлежащее этому пакету. Это весьма соответствует стилю Perl.

Объявления переменных с областью видимости

Оставшаяся часть главы посвящена в основном использованию глобальных переменных. Или скорее тому, как *не* использовать глобальные переменные. Есть несколько объявлений, помогающих обойтись без глобальных переменных – или, по крайней мере, без неразумного их применения.

Мы уже упоминали объявление пакета, давно введенное в Perl для того, чтобы позволить разбить глобальные переменные на несколько пакетов. Для некоторых типов переменных это действует довольно хорошо. Пакеты используются библиотеками, модулями и классами для хранения данных своих интерфейсов (и некоторых своих полузакрытых данных), чтобы избежать конфликтов с одноименными переменными и функциями в основной программе или других модулях. Так, запись `$Some::stuff`[2] означает, что используется скалярная переменная `$stuff` из пакета `Some`. См. главу 10 «Пакеты».

Если бы в этом заключалось все, относящееся к данному вопросу, то программы Perl по мере увеличения в размере быстро становились бы громоздкими. К счастью, имеющиеся в Perl три объявления области видимости позволяют легко создавать совершенно закрытые переменные (с помощью `my`), предоставлять избирательный доступ к глобальным переменным (посредством `our`) и обеспечивать временные значения для глобальных переменных (с использованием `local`):

---

[1] А также неквалифицированные имена подпрограмм, дескрипторов файлов и форматов.

[2] Или архаическую `$Some'stuff`, использование которой за рамками поэзии на Perl не приветствуется.

```
my $nose;
our $House;
local $TV_channel;
```

Если перечисляется более одной переменной, то список должен быть заключен в круглые скобки. Для my и our элементы могут быть только простыми скалярными переменными, переменными массивов и хешей. Для local ограничения несколько более слабые: можно сделать локальными также целые записи таблицы имен typeglob и отдельные элементы или срезы массивов и хешей:

```
my ($nose, @eyes, %teeth);
our ($House, @Autos, %Kids);
local (*Spouse, $phone{HOME});
```

Каждый из этих модификаторов предоставляет свой тип «заключения» для модифицируемых переменных. Несколько упрощая, можно сказать, что our ограничивает область видимости имен, local ограничивает область видимости значений, а my ограничивает область видимости как имен, так и значений.

Для каждой из этих конструкций может быть выполнено присваивание, хотя они различаются по тому, что фактически осуществляют со значениями, поскольку имеют различные механизмы хранения значений. Они также несколько различаются, если вы *не* присваиваете (как было сделано выше) им значений: объявления my и local влечет то, что начальными значениями модифицируемых переменных становятся, в зависимости от типа, undef или (), тогда как our оставляет текущее значение соответствующей глобальной переменной неизменным.

Синтаксически my, our и local являются просто модификаторами (как прилагательные) выражения l-значения. Когда выполняется присваивание модифицированному l-значению, модификатор не влияет на то, рассматривается ли l-значение как скаляр или как список. Чтобы узнать, как будет работать присваивание, можно просто представить себе, что модификатор отсутствует. Поэтому каждое из:

```
my ($foo) = <STDIN>;
my @array = <STDIN>;
```

предоставляет списковый контекст для правой части, в то время как:

```
my $foo = <STDIN>;
```

предоставляет скалярный контекст.

Модификаторы связывают аргументы более прочно (с более высоким приоритетом), чем это делает запятая. В следующем примере ошибочно модифицируется одна, а не две переменных, поскольку список, следующий за модификатором, не заключен в круглые скобки.

```
my $foo, $bar = 1;              # НЕВЕРНО
```

Это равносильно следующему:

```
my $foo;
$bar = 1;
```

Если включен вывод предупредительных сообщений с помощью ключей командной строки -w или -W или, что предпочтительней, через объявление use warnings, описываемое далее в разделе «Прагмы», то будет получено сообщение об ошибке.

В целом, лучше всего объявить для переменной минимальную допустимую область видимости, которая годится для нее. Поскольку переменные, объявляемые в операторе управления логикой программы, видимы только в блоке, которым управляет этот оператор, их видимость является ограниченной. Читаемость на естественном языке благодаря этому тоже улучшается.

```
sub check_warehouse {
    for my $widget (our @Current_Inventory) {
        print "Сегодня у меня на складе есть $widget.\n";
    }
}
```

Чаще всего встречается тип объявления my, объявляющий переменные с лексической областью видимости, имена и значения которых хранятся во временной памяти текущей области видимости и не доступны глобально. Очень близко к нему объявление our, которое вводит в текущую область видимости имя с лексической областью видимости, как и my, но при этом фактически ссылается на глобальную переменную, к которой при желании может обратиться каждый. Иными словами, это глобальная переменная, маскирующаяся под лексическую.

Другая форма области видимости, динамическая область видимости, применяется к переменным с модификатором local, которые, несмотря на слово «local», являются на самом деле глобальными переменными и не имеют никакого отношения к локальной временной памяти.

Переменные с лексической областью видимости: my

Чтобы помочь вам избежать хлопот, связанных с поддержкой глобальных переменных, Perl предоставляет переменные с лексической областью видимости, часто для краткости называемые *лексическими* (*lexicals*). В отличие от глобальных переменных, лексические гарантируют приватность. Конечно, если ссылки на эти переменные не выставляются напоказ, что позволило бы манипулировать ими косвенным образом, можете быть уверены, что всякий доступ к этим закрытым переменным ограничен кодом в одной отдельной и легко идентифицируемой части вашей программы. В конце концов, потому мы и выбрали ключевое слово my (мой).

В последовательности операторов могут содержаться объявления переменных с лексической областью видимости. Такие объявления обычно помещают

впереди последовательности операторов, но это не обязательное требование. Помимо объявления имен переменных на этапе компиляции, эти объявления действуют как обычные операторы на этапе исполнения: все они обрабатываются внутри последовательности операторов, как если бы были обычными операторами без модификаторов:

```
my $name = "fred";
my @stuff = ("car", "house", "club");
my ($vehicle, $home, $tool) = @stuff;
```

Эти лексические переменные совершенно скрыты от всего мира за пределами непосредственно охватывающей их области видимости. В отличие от действия динамической области видимости, создаваемой `local` (см. следующий раздел), лексические переменные скрыты от любых подпрограмм, вызываемых из своей области видимости. Это верно даже в отношении той же самой подпрограммы, вызываемой из себя самой или из любого другого места: каждый экземпляр подпрограммы получает собственную «временную память» для лексических переменных.

В отличие от области видимости блока, области видимости файла не могут быть вложенными; «охват», по крайней мере, текстуально, не имеет места. Если вы загружаете код из отдельного файла с помощью `do`, `require` или `use`, то код в этом файле не имеет доступа к вашим лексическим переменным, равно как и вы не имеете доступа к лексическим переменным из этого файла.

Однако любая область видимости внутри файла (и даже сам файл) действует по правилам. Часто удобно иметь области видимости более широкие, чем определения функций, так как это позволяет организовать совместный доступ к закрытым переменным для ограниченного набора подпрограмм. Так создаются переменные, которые С-программист назвал бы статическими:

```
{
    my $state = 0;
    sub on     { $state = 1 }
    sub off    { $state = 0 }
    sub toggle { $state = !$state }
}
```

Оператор `eval` *`STRING`* тоже действует как вложенная область видимости, поскольку код в `eval` может видеть лексические переменные вызвавшей его области (если их имена не замаскированы идентичными объявлениями в собственной области видимости `eval`). Аналогично анонимные подпрограммы могут обращаться ко всем лексическим переменным охватывающей их области видимости; когда это происходит, их называют *замыканиями* (*closures*).[1] Объединим эти два замечания: если блок выполняет `eval` над строкой

[1] Мнемоническое правило основывается на общем элементе в «en*clos*ing scope» (охватывающая область видимости) и «*clos*ure» (замыкание). (В действительности определение замыкания происходит от математического понятия, означающего полноту набора величин и операций над этими величинами.)

и создает при этом анонимную подпрограмму, эта подпрограмма становится замыканием с полным доступом к лексическим переменным как eval, так и блока, даже после выхода из eval и блока. См. раздел «Замыкания» главы 8 «Ссылки».

Вновь объявленная переменная (или значение в случае local) не видна до оператора, следующего *после* оператора, содержащего объявление. Поэтому можно копировать переменную так:

```
my $x = $x;
```

При этом новая внутренняя переменная $x инициализируется текущим значением $x независимо от того, является ли текущая $x глобальной или лексической. (Если новая переменная не инициализируется, ее начальным значением будет неопределенное или пустое значение.)

Объявление лексической переменной с некоторым именем маскирует все ранее объявленные лексические переменные с таким же именем. При этом маскируются также неквалифицированные глобальные переменные с тем же именем, но к глобальной переменной, однако, всегда можно получить доступ, явно квалифицировав ее именем пакета, в котором она находится, например $PackageName::varname.

Глобальные объявления с лексической областью видимости: our

Более правильным способом доступа к глобальным переменным, особенно из программ и модулей с объявлением use strict, является объявление our. Это объявление имеет лексическую область видимости, поскольку действует только до конца текущей области видимости. Но, в отличие от лексической области видимости my или динамической области видимости local, our не изолирует ничего в текущей лексической или динамической области видимости. Напротив, оно обеспечивает доступ к глобальной переменной в текущем пакете, маскируя одноименные лексические переменные, которые, в противном случае, закрыли бы от вас эту глобальную переменную. В этом отношении переменные our действуют точно так же, как переменные my.

Если поместить объявление our за пределами всех блоков, заключенных в фигурные скобки, то оно действует до конца текущей компилируемой единицы. Однако часто его помещают вверху определения подпрограммы, чтобы показать, что обращаются к глобальной переменной:

```
sub check_warehouse {
    our @Current_Inventory;
    my  $widget;
    foreach $widget (@Current_Inventory) {
        print "Сегодня у меня на складе есть $widget.\n";
    }
}
```

Поскольку глобальные переменные живут дольше и видимы шире, чем закрытые, мы предпочитаем использовать для них более длинные и броские имена, чем для временных переменных. Одно лишь это правило, если ему старательно следовать, может столь же успешно отбить охоту к работе с глобальными переменными, как и использование `use strict`, особенно у тех, кто не особенно искусен в обращении с клавиатурой.

Повторные объявления `our` не имеют смысла вложенности. Каждое вложенное `my` создает новую переменную, а каждое вложенное `local` создает новое значение. Но всякий раз, когда используется `our`, речь идет о *той же самой* глобальной переменной без учета вложенности. Если осуществляется присвоение переменной `our`, его результат сохраняет свое действие и после конца области видимости объявления. Дело в том, что `our` не создает значений; оно просто предоставляет ограниченную форму доступа к глобальной переменной, срок жизни которой не ограничен:

```
our $PROGRAM_NAME = "waiter";
{
    our $PROGRAM_NAME = "server";
    # Вызываемый здесь код видит "server".
    ...
}
# Выполняемый здесь код по-прежнему видит "server".
```

Сравните все это с тем, что происходит с `my` или `local`, когда за пределами блока наружная переменная или значение снова становятся видимыми:

```
my $i = 10;
{
    my $i = 99;
    ...
}
# Компилируемый здесь код видит наружную переменную.
local $PROGRAM_NAME = "waiter";
{
    local $PROGRAM_NAME = "server";
    # Код, выполняемый здесь, видит "server".
    ...
}
# Код, выполняемый здесь, снова видит "waiter".
```

Обычно имеет смысл делать только одно присваивание в объявлении `our`, вероятно, в самом верху программы или модуля, либо, реже, когда вы предваряете `our` собственным `local`:

```
{
    local our @Current_Inventory = qw(bananas);
    check_warehouse();  # нет, бананов у нас нет :-)
}
```

Переменные с динамической областью видимости: local

Использование объявления `local` с глобальной переменной дает ей временное значение при каждом выполнении `local`, но не оказывает влияния на глобальную видимость переменной. Когда программа достигает конца этой динамической области видимости, временное значение отбрасывается и восстанавливается первоначальное значение. Но эта переменная всегда по-прежнему остается глобальной, лишь получающей временное значение, пока выполняется этот блок. Если вызвать какую-либо другую функцию в то время, когда эта глобальная переменная содержит временное значение, и эта функция обратится к данной глобальной переменной, то она увидит ее временное значение, а не первоначальное. Иными словами, эта другая функция находится в вашей динамической области видимости, хотя, возможно, не в лексической области видимости.[1]

Если вы встречаете объявление `local`, которое выглядит так:

```
{
    local $var = $newvalue;
    some_func();
    ...
}
```

то можете представлять его себе исключительно в терминах присваиваний этапа выполнения:

```
{
    $oldvalue = $var;
    $var = $newvalue;
    some_func();
    ...
}
continue {
    $var = $oldvalue;
}
```

Разница в том, что при использовании `local` значение восстанавливается независимо от того, каким образом происходит выход из блока, даже если это досрочный возврат из данной области видимости. Переменная остается все той же глобальной переменной, но обнаруживаемое в ней значение зависит от того, из какой области видимости вызвана функция. Вот почему эта область видимости называется *динамической*: она изменяется на этапе исполнения.

---

[1] Вот почему лексические области видимости иногда называют *статическими*: чтобы противопоставить их динамическим областям видимости и подчеркнуть их детерминированность на этапе компиляции. Не путайте это значение термина с тем, которое слово «статический» имеет в C и C++. Этот термин слишком сильно перегружен, почему мы и избегаем его.

Как и в случае применения my, можно инициализировать local копией той же самой глобальной переменной. Все изменения, происходящие с этой переменной во время выполнения подпрограммы (и всех других, вызываемых из нее, которые, разумеется, по-прежнему видят глобальную переменную с динамической областью видимости), будут отброшены при возврате из подпрограммы. Конечно, лучше всего снабдить комментариями свои действия:

```
# ПРЕДУПРЕЖДЕНИЕ: Изменения в этой динамической области видимости являются
временными.
local $Some_Global = $Some_Global;
```

После этого глобальная переменная по-прежнему видна всюду в программе независимо от того, была ли она явно определена с помощью our или возникла из ничего, или содержит значение local, подлежащее отбрасыванию при выходе из области видимости. В маленьких программках это не столь плохо, но в больших программах вы быстро потеряете возможность отслеживать, где в коде применяются все эти глобальные переменные. При желании можно запретить случайное использование глобальных переменных с помощью директивы use strict 'vars', описываемой в следующем разделе.

Хотя и my и local предоставляют некоторую степень защиты, в целом предпочтительнее использовать my, а не local. Однако иногда применение local необходимо, чтобы временно изменить значение существующих глобальных переменных, например, перечисленных в главе 28 «Специальные имена». Лексическую область видимости могут иметь только буквенно-цифровые идентификаторы, а многие из этих специальных переменных не являются строго буквенно-цифровыми. Без local также не обойтись при осуществлении временных изменений в таблице имен пакета, как показано в разделе «Таблицы имен» главы 10 «Пакеты». Наконец, можно распространить действие local на один элемент или целый срез массива или хеша. Это допустимо, даже если массив или хеш являются лексической переменной, при этом режим динамической области видимости local накладывается поверх этих лексических переменных. Мы не станем здесь больше распространяться о семантике local. Дополнительные сведения можно получить из описания local в главе 29 «Функции».

Прагмы

Многие языки программирования позволяют давать компилятору советы. В Perl эти советы сообщаются компилятору посредством прагмы (директивы) use. Вот некоторые из них:

```
use warnings;
use strict;
use integer;
use bytes;
use constant pi => ( 4 * atan2(1,1) );
```

Все прагмы Perl описаны в главе 31 «Модули прагм», но сейчас мы отдельно поговорим о некоторых из них, особенно полезных в связи с материалом, излагаемым в данной главе.

Хотя некоторые директивы представляют из себя глобальные объявления, воздействующие на глобальные переменные в текущем пакете, в большинстве своем это объявления с лексической областью видимости, действие которых продолжается только до конца охватывающего блока, файла или `eval` (в зависимости от того, что встретится раньше). Директива с лексической областью видимости может быть отменена во внутренней области видимости с помощью объявления `no`, которая действует как `use`, но противоположным образом.

Управление выводом предупредительных сообщений

Чтобы показать, как это работает, мы поработаем с директивой `warnings`, указывая Perl, нужно ли выводить предупредительные сообщения по поводу сомнительных приемов программирования:

```
use warnings;       # Включить вывод предупреждений отсюда и до конца файла.
...
{
    no warnings;    # Отключить вывод предупреждений до конца блока.
    ...
}
# Здесь вывод предупреждений автоматически снова активизируется.
```

Когда вывод предупредительных сообщений включен, Perl сообщает о переменных, используемых только один раз, объявлениях переменных, маскирующих другие объявления в той же области видимости, недопустимых преобразованиях строк в числа, использовании неопределенных значений в качестве законных строк или чисел, попытках записи в файлы, открытые только для чтения (или вообще не открытые), и многих других ситуациях, описанных в главе 33 «Диагностические сообщения».

Прагма `use warnings` является предпочтительным способом управления выводом предупредительных сообщений. В старых программах можно было только использовать ключ командной строки `-w` или изменять глобальную переменную `$^W`:

```
{
    local $^W = 0;
    ...
}
```

Гораздо лучше применять директивы `use warnings` и `no warnings`. Лучше, потому что это происходит на этапе компиляции, представляет собой лексическое объявление и, следовательно, не может влиять на код, на который не должно влиять, и потому (хотя мы не показали этого в наших простых примерах), что позволяет тонко управлять отдельными классами сообщений. Дополнительные сведения о директиве `warnings`, включая и то, как преобра-

зовать просто надоедливые сообщения в фатальные ошибки, и как переопределить директиву, чтобы включить глобальный вывод предупредительных сообщений, даже если модуль требует обратного, можно найти в описании `use warnings` в главе 31 «Модули прагм».

Управление использованием глобальных переменных

Еще одно часто встречающееся объявление – это прагма `use strict`, выполняющая несколько функций, одной из которых является управление использованием глобальных переменных. Обычно Perl позволяет создавать новые переменные (а нередко и «затаптывать» старые) путем простого их упоминания. Никаких объявлений переменных не требуется – имеется в виду по умолчанию. Поскольку бесконтрольное применение глобальных переменных может сделать мучительным сопровождение больших программ и модулей, иногда желательно воспрепятствовать случайному их использованию. Чтобы содействовать предотвращению таких несчастных случаев, можно сказать:

```
use strict 'vars';
```

Это означает, что с данного места и до конца охватывающей области видимости все упоминаемые переменные должны относиться либо к лексическим переменным, либо к явным образом разрешенным глобальным, в противном случае возникает ошибка компиляции. Глобальная переменная считается явно разрешенной в одном из следующих случаев:

- Это одна из специальных переменных Perl, действующих всюду в программе (см. главу 28 «Специальные имена»).
- Она полностью квалифицирована именем своего пакета (см. главу 10 «Пакеты»).
- Она импортирована в текущий пакет (см. главу 11 «Модули»).
- Она маскирует переменную с лексической областью видимости посредством объявления `our`. (Это основная причина, по которой мы ввели в Perl объявления `our`.)
- Конечно, всегда есть пятая альтернатива: если применение директивы становится обременительным, можно просто отменить ее во внутреннем блоке с помощью

  ```
  no strict 'vars'
  ```

С помощью этой директивы можно также включить строгую проверку символического разыменования и случайного использования простых слов. Обычно просто говорят:

```
use strict;
```

чтобы включить все три ограничения. Дополнительные сведения можно найти в описании `use strict` в главе 31 «Модули прагм».

5

Поиск по шаблону

Встроенная в Perl поддержка поиска по шаблону обеспечивает удобный и эффективный поиск в больших объемах данных. Perl превосходно подойдет тем, кто управляет большим коммерческим порталом в Сети и просматривает все поступающие новости в поисках интересных данных, правительственным организациям, занимающимся анализом демографических данных (или генома человека), или тем, кто работает в образовательных учреждениях и пытается организовать представление каких-то динамических данных на своем веб-сайте, отчасти благодаря его возможностям работы с базами данных, но в большей степени благодаря возможностям поиска по шаблону. Если рассматривать «текст» в самом широком смысле этого слова, то, вероятно, 90% всего, что мы делаем, на 90% состоит из обработки текста. Это действительно та область, на которую нацелен Perl, и всегда был нацелен; эта цель даже присутствует в самом названии Perl: Practical *Extraction* and Report Language – практический язык *извлечения* данных и генерации отчетов. Шаблоны Perl предоставляют мощный способ просмотра гор простых данных и извлечения из них полезной информации.

Шаблон задается путем создания *регулярного выражения* (*regular expression* или *regex*). Машина регулярных выражений Perl («Engine») берет это выражение и определяет, соответствует ли этот шаблон (и каким образом) вашим данным. Хотя большая часть данных представляет собой, вероятно, текстовые строки, ничто не мешает применять регулярные выражения для поиска и замены в любых последовательностях байт, даже если обычно мы рассматриваем их как «двоичные» данные. Для Perl байты – это просто символы,

порядковые значения которых оказались меньшими 256. (Подробнее об этом см. главу 15 «Unicode».)

Тех, кто уже сталкивался где-то с регулярными выражениями, мы должны предупредить, что в Perl регулярные выражения несколько иные. Во-первых, они не являются полностью регулярными в теоретическом смысле слова, поскольку их возможности значительно шире, чем у традиционных регулярных выражений, которые преподаются в курсах вычислительной техники. Во-вторых, они так часто применяются в Perl, что имеют свои особые переменные, операторы и соглашения по использованию кавычек, тесно интегрированные с языком, а не просто свободно прикреплены, как любая другая библиотека. Программисты, не имеющие опыта работы с Perl, нередко тщетно ищут функции для поиска и замены вроде

```
match( $строка, $шаблон );
subst( $строка, $шаблон, $замена );
```

Но поиск и подстановка являются в Perl такими фундаментальными задачами, что заслужили для себя однобуквенные операторы: `m/PATTERN/` (/шаблон/) и `s/PATTERN/REPLACEMENT/` (/шаблон/замена/), для краткости – `m//` и `s///`. Они не только имеют краткий синтаксис, но и анализируются не как обычные операторы, а как строки в двойных кавычках; тем не менее, они действуют как операторы, и так мы и будем их называть. На протяжении всей этой главы рассматривается применение этих операторов для поиска шаблонов в строке. Если какая-то часть строки согласуется с шаблоном, то мы будем говорить, что поиск успешен. Если поиск успешен, то можно выполнить массу мощных операций. В частности, в случае применения `s///` успешный поиск приводит к замене обнаруженной части строки тем, что было задано в качестве *REPLACEMENT*.

Вся эта глава посвящена построению и использованию шаблонов. Регулярные выражения Perl представляют собой мощной средство, позволяющее уместить много смысла в малом объеме. Поэтому они могут выглядеть устрашающе для того, кто попытается догадаться, что же в целом делает некий длинный шаблон. Но если суметь разбить его на части и знать, как машина эти части интерпретирует, то можно понять действие любого регулярного выражения. Нередко можно видеть, как сотня строк программы на C или Java выражается на Perl регулярным выражением, состоящим из одной строки. Понять это регулярное выражение может оказаться несколько труднее, чем каждую отдельную строку длинной программы, но, с другой стороны, регулярное выражение, возможно, будет значительно легче понять, чем длинную программу в целом. Просто имейте такие вещи в виду.

Бестиарий регулярных выражений

Прежде чем погрузиться в изучение правил интерпретации регулярных выражений, посмотрим, как выглядят некоторые шаблоны. Большинство символов в регулярных выражениях просто соответствуют самим себе. Если

несколько символов выстроены подряд, они должны быть найдены, как и можно ожидать, в указанном порядке. Поэтому, если записать шаблон для поиска:

```
/Frodo/
```

то можно быть уверенным, что поиск будет успешным, только если строка содержит в каком-то месте подстроку "Frodo". (*Подстрока* является просто частью строки.) Соответствие может находиться в любом месте строки, если только в нем рядом расположены эти пять символов в указанном порядке.

Другие символы не соответствуют самим себе, а некоторым образом «неправильно» ведут себя. Мы называем их *метасимволами*. (Все метасимволы ведут себя неправильно сами, но некоторые из них настолько дурны, что «заражают» плохим поведением и соседние символы.)

Вот эти негодяи:

```
\ | ( ) [ { ^ $ * + ? .
```

Метасимволы на практике очень полезны и имеют в шаблонах особое значение; по ходу дела мы об этих значениях расскажем. Но хотим уверить читателя, что всегда можно осуществить поиск этих символов литерально, помещая перед ними обратную косую черту (обратный слэш). Например, обратная косая черта сама является метасимволом, поэтому для буквального поиска обратной косой черты нужно поместить обратную косую черту перед ней самой: \\.

Как видите, обратный слэш – это один из тех символов, которые провоцируют неправильное поведение других символов. Результатом того, что мы заставляем плохо себя вести плохо ведущий себя метасимвол, является его хорошее поведение – как если бы это было двойное отрицание. Поэтому помещение обратной косой черты перед символом для того, чтобы он воспринимался литерально, действует, но только на символы пунктуации; обратная косая черта перед (обычно хорошо воспитанным) буквенно-цифровым символом производит противоположный эффект: она превращает буквальный символ в нечто особое. Встретив такие двухсимвольные последовательности:

```
\b \D \t \3 \s
```

необходимо знать, что эта последовательность является *метасимволом*, соответствующим чему-то необычному. Например, \b соответствует границе слова, а \t – обычному символу табуляции. Обратите внимание, что табуляция имеет размер в один символ, тогда как граница слова имеет ширину ноль символов, поскольку это место между двумя символами. Поэтому мы называем \b утверждением *нулевой ширины* (*zero-width* assertion). Тем не менее, \t и \b схожи в том, что оба утверждают нечто о конкретном месте в строке. *Утверждая* что-либо в регулярном выражении, мы просто требуем, чтобы это нечто было истинным при соответствии шаблону.

Большинство участков регулярного выражения представляют собой утверждения того или иного вида, при этом обычные символы просто утверждают, что они соответствуют самим себе. Если быть точным, то они утверждают также, что нечто *следующее* будет иметь соответствие на один символ дальше в строке, поэтому мы и говорим, что символ табуляции «имеет ширину в один символ». Некоторые утверждения (такие, как `\t`) «съедают» в случае своего соответствия часть строки, в то время как другие (такие, как `\b`) не делают этого. Но обычно мы используем термин «утверждение» только для утверждений нулевой ширины. Чтобы избежать путаницы, мы будем называть нечто, обладающее шириной, *атомом*. (Если вы физик, можете представлять себе атомы с ненулевой шириной как имеющие массу, в противоположность утверждениям нулевой ширины, которые не имеют массы, как фотоны.)

Мы увидим также, что некоторые метасимволы не являются утверждениями, а выполняют функцию структурирования (подобно тому, как фигурные скобки и точки с запятой определяют структуру обычного кода Perl, но не совершают никаких действий). Эти структурирующие метасимволы в некоторых отношениях являются самыми важными, поскольку при обучении чтению регулярных выражений решающий первый шаг состоит в том, чтобы приучить глаз выхватывать структурирующие метасимволы. После освоения этой премудрости чтение регулярных выражений становится легким, как дуновение ветерка.[1]

Один из таких структурирующих символов – вертикальная черта, обозначающая *чередование* (*alternation*):

```
/Frodo|Pippin|Merry|Sam/
```

Это означает, что нахождение любой из этих строк делает поиск успешным; об этом рассказывается в разделе «Чередование» далее в этой главе. А в следующем за ним разделе «Захват и кластеризация» мы покажем, как осуществлять *группирование* путем заключения частей шаблона в круглые скобки:

```
/(Frodo|Drogo|Bilbo) Baggins/
```

или даже:

```
/(Frod|Drog|Bilb)o Baggins/
```

Еще вы увидите то, что называется *квантификаторами*, они указывают, сколько раз подряд должно быть найдено то, что им предшествует. Квантификаторы выглядят так:

```
*  +  ?  *?  {3}  {2,5}
```

[1] Временами, положим, ветерка довольно свежего, но не настолько, чтобы свалить с ног.

Однако в такой изоляции вы их не увидите. Квантификаторы имеют смысл только тогда, когда прикреплены к атомам, т. е. утверждениям, имеющим ширину.[1] Квантификаторы прикрепляются только к предшествующему атому; на человеческом языке это означает, что обычно они указывают количество повторений только одного символа. Для того чтобы найти три экземпляра "bar" подряд, нужно сгруппировать отдельные символы "bar" в одной «молекуле» с помощью скобок, вот так:

```
/(bar){3}/
```

Это будет соответствовать строке "barbarbar". Если бы вы сказали /bar{3}/, это соответствовало бы "barrr". Подробнее о квантификаторах см. далее раздел «Квантификаторы».

Итак, вы посмотрели на некоторых бестий, обитающих в регулярных выражениях, и вам, вероятно, захотелось приступить к их приручению. Однако прежде чем заняться серьезным обсуждением регулярных выражений, надо вернуться немного назад и поговорить об операторах поиска по шаблону, использующих регулярные выражения. (И если по дороге вам случится обнаружить еще несколько зверюшек из регулярных выражений, оставьте приличные чаевые своему гиду.)

Операторы поиска по шаблону

Рассуждая в терминах зоосада, можно сказать, что операторы Perl поиска по шаблону действуют как своего рода клетки для регулярных выражений, из которых последние не должны выбраться. Дело в том, что если выпустить зверюшек из семейства регулярных выражений на просторы языка, Perl может превратиться в настоящие джунгли. Конечно, миру нужны джунгли – в конце концов, они поддерживают биологическое многообразие – но джунгли должны находиться там, где им положено. Аналогично, несмотря на то что они являются двигателем комбинаторного многообразия, регулярные выражения должны оставаться внутри операторов поиска по шаблону, где их место. Там находятся джунгли.

Как если бы мощи регулярных выражений было недостаточно, операторы m// и s/// предоставляют также (тоже ограниченную по месту) мощь интерполяции строк в двойных кавычках. Поскольку шаблоны анализируются как строки в двойных кавычках, действуют все обычные соглашения для двойных кавычек, в том числе интерполяция переменных (если только в качестве ограничителя не используются одинарные кавычки) и специальные сим-

[1] Квантификаторы немного похожи на модификаторы операторов, описанные в главе 4 «Операторы и объявления», которые могут лишь прикрепляться к отдельным операторам. Прикрепление квантификатора к утверждению нулевой длины было бы похоже на попытку прикрепить модификатор while к объявлению – и то и другое имело бы не более смысла, чем попросить у аптекаря фунт фотонов. Аптекари покупают и продают только атомы и подобные им вещи.

волы, обозначаемые с помощью обратной косой черты. (См. раздел «Специальные символы» далее в этой главе.) Они применяются раньше, чем строка интерпретируется как регулярное выражение. (Это один из немногочисленных случаев в Perl, когда строка обрабатывается более чем за один проход.) Первый проход представляет собой не совсем обычную интерполяцию строки в двойных кавычках, поскольку известно, что на нем должно интерполироваться и что должно быть передано анализатору регулярных выражений. Поэтому, например, символ $, сразу за которым следует вертикальная черта, закрывающая круглая скобка или конец строки, будет обрабатываться не как интерполяция переменной, а как обычное утверждение регулярного выражения, означающее конец строки. Поэтому, если сказать:

```
$foo = "bar";
/$foo$/;
```

то на проходе интерполяции в двойных кавычках известно, что эти два символа $ действуют по-разному. Поэтому производится интерполяция $foo, а затем анализатору регулярных выражений передается следующее:

```
/bar$/;
```

Другим следствием этого двухпроходного анализа является то, что обычный анализатор лексем Perl сначала находит конец регулярного выражения, совсем как если бы он искал завершающий ограничитель для обычной строки. Только после того, как найден конец строки (и завершена интерполяция всех переменных), шаблон рассматривается как регулярное выражение. Помимо всего прочего это означает, что нельзя «скрыть» завершающий ограничитель внутри конструкции регулярного выражения (такой как класс символов или комментарий в регулярном выражении, о которых мы еще расскажем). Perl увидит ограничитель, где бы он ни был, и закончит шаблон в этом месте.

Следует также знать, что интерполяция переменных в шаблоне замедляет работу механизма поиска по шаблону, поскольку он считает, что нужно проверить, не изменилась ли переменная, если ему приходится заново компилировать шаблон (что еще более замедляет работу). См. раздел «Интерполяция переменных» далее в этой главе.

Оператор транслитерации[1] tr/// не осуществляет интерполяцию переменных, он даже не использует регулярные выражения! (На самом деле ему, возможно, вообще не место в этой главе, но лучшего для него мы не нашли.) Однако у него есть одна общая черта с m// и s///: он выполняет привязку к переменным с помощью операторов =~ и !~.

Операторы =~ и !~, описанные в главе 3 «Унарные и бинарные операторы», привязывают скалярное выражение из своей левой части к одному из трех

[1] transliteration operator – обычно переводят как «оператор замены», но термин «транслитерация» в данном случае точнее отражает суть происходящего, т. к. указывает на побуквенную замену. – *Примеч. перев.*

операторов типа кавычек, находящихся в их правой части: m// для поиска по шаблону, s/// для подстановки некоторой строки вместо подстроки, отыскиваемой по шаблону, и tr/// (или его синоним – y///) для транслитерации (замены) одного набора символа другим. (Можно также записывать m// как //, без m, если в качестве ограничителя выступает косая черта.) Если в правой части =~ или !~ находится нечто отличное от этих трех операторов, оно все равно рассматривается как операция поиска m//, но при этом не остается места для каких-либо замыкающих модификаторов (см. далее раздел «Модификаторы шаблонов»), и вам придется обрабатывать свои собственные кавычки:

```
print "соответствует" if $somestring =~ $somepattern;
```

На самом деле нет оснований, чтобы не написать это явно:

```
print "соответствует" if $somestring =~ m/$somepattern/;
```

При использовании в операции поиска =~ и !~ иногда читаются, как «подходит» (matches) и «не подходит», соответственно (хотя «содержит» и «не содержит» способствовали бы меньшей путанице).

Кроме операторов – m// и s/// – регулярные выражения появляются в Perl еще в двух местах. Первый аргумент функции split представляет собой особый оператор поиска, определяющий, что *не* должно возвращаться при разбиении строки на несколько подстрок. См. описание и примеры применения split в главе 29 «Функции». Оператор qr// («quote regex» – заключить в кавычки регулярное выражение) тоже задает шаблон посредством регулярного выражения, но он не пытается ничего найти (в отличие от m//, который пытается). Вместо этого он возвращает скомпилированную форму регулярного выражения, которую можно будет использовать в дальнейшем. Дополнительные сведения можно найти в разделе «Интерполяция переменных» данной главы.

Один из операторов m//, s/// или tr/// может быть применен к конкретной строке с помощью оператора привязки =~ (который, в сущности, является не оператором, а некоторым «тематизатором», рассуждая в терминах лингвистики). Вот несколько примеров:

```
$haystack =~ m/needle/                    # поиск по простому шаблону
$haystack =~  /needle/                    # то же

$italiano =~ s/butter/olive oil/          # полезная для здоровья замена

$rotate13 =~ tr/a-zA-Z/n-za-mN-ZA-M/      # простое (для раскрытия)
                                          #  шифрование
```

В отсутствие оператора привязки в качестве «темы» неявным образом используется $_:

```
/new life/ and            # искать в $_ и (если найдена)
    /new civilizations/   #  смело снова искать в $_
```

```
s/sugar/aspartame/          # заменить на заменитель в $_

tr/ATCG/TAGC/               # дополнить нить ДНК в $_
```

Поскольку s/// и tr/// модифицируют скаляры, к которым применяются, их можно использовать только с допустимыми левыми значениями:

```
"onshore" =~ s/on/off/;      # НЕВЕРНО: ошибка этапа компиляции
```

Однако m// работает с результатом любого скалярного выражения:

```
if ((lc $magic_hat->fetch_contents->as_string) =~ /rabbit/) {
    print "Эй, что тут происходит?\n";
}
else {
    print "Фокус не удался!\n";
}
```

Но следует проявлять некоторую осторожность, поскольку =~ и !~ имеют достаточно высокий приоритет: в нашем предшествующем примере необходимо заключить левый терм в круглые скобки.[1] Оператор привязки !~ действует подобно =~, но отрицает логический результат операции:

```
if ($song !~ /words/) {
    print qq/Похоже, что"$song" является песней без слов.\n/;
}
```

Поскольку m//, s/// и tr/// являются операторами кавычек, мы можем выбрать собственные ограничители. Они действуют так же, как операторы кавычек q//, qq//, qr// и qw// (см. раздел «Выберите собственные кавычки» в главе 2 «Всякая всячина»).

```
$path =~ s#/tmp#/var/tmp/scratch#;

if ($dir =~ m[/bin]) {
    print "Пожалуйста, без каталогов с двоичными модулями.\n";
}
```

Если в s/// или tr/// используются парные ограничители, и пара в первой части является одной из обычных четырех (угловые, круглые, квадратные или фигурные), то для второй части можно выбрать ограничители, отличающиеся от выбранных для первой:

```
s(egg)<larva>;
s{larva}{pupa};
s[pupa]/imago/;
```

[1] В отсутствие скобок имеющая более низкий приоритет lc была бы применена ко всему шаблону поиска, а не только к вызову метода объекта «волшебный_цилиндр».

Перед открывающими ограничителями допустимы пробельные символы:

```
s (egg)   <larva>;
s {larva} {pupa};
s [pupa]  /imago/;
```

Всякий раз, когда поиск по шаблону оказывается успешным (в том числе при замене по шаблону), переменные $`, $& и $' устанавливаются равными тексту слева от найденного соответствия, всему соответствию и тексту справа от соответствия. Это удобно для разделения строк на части:

```
"hot cross buns" =~ /cross/;
print "Найдено: <$`> $& <$'>\n";          # Найдено: <hot > cross < buns>
print "Слева:        <$`>\n";             # Слева:        <hot >
print "Соответствие: <$&>\n";             # Соответствие: <cross>
print "Справа:       <$'>\n";             # Справа:       < buns>
```

Детализацию и эффективность можно увеличить при помощи скобок, позволяющих захватывать отдельные участки того, что требуется сохранить. Каждая пара круглых скобок захватывает подстроку, соответствующую *подшаблону*, заключенному в эти скобки. Пары скобок нумеруются слева направо в соответствии с позицией левых скобок; подстроки, соответствующие этим субшаблонам, после нахождения соответствия доступны как пронумерованные переменные $1, $2, $3 и т. д.:[1]

```
$_ = "Бильбо Бэггинс родился 22 сентября";
/(.*) родился (.*)/;
print "Имя: $1\n";
print "Дата: $2\n";
```

$`, $&, $' и нумерованные переменные являются глобальными переменными, неявно локализованными в охватывающей динамической области видимости. Их значение сохраняется до очередного успешного поиска по шаблону или конца текущей области видимости, смотря по тому, что произойдет раньше. Подробнее об этом потом, в другой области видимости.

Когда Perl видит, что где-то в программе нужна одна из переменных, $`, $& или $', он создает их при каждом поиске по шаблону. Это несколько замедляет выполнение программы. Аналогичный механизм используется в Perl для создания переменных $1, $2 и т. д., поэтому приходится платить и за каждый шаблон, содержащий захватывающие скобки. (См. в разделе «Кластеризация», как избежать издержек захвата при сохранении режима группирования.) Но если ни разу не использовать $`, $& или $', то шаблоны *без* захватывающих скобок штрафом не облагаются. Поэтому обычно лучше избегать применения $`, $& и $', если есть такая возможность, особенно в библиотечных модулях. Но если вы хоть раз их использовали (а в некоторых алгоритмах это действительно удобно), то можете дальше уже не особенно

[1] Не считая $0, в которой находится имя программы.

стесняться, поскольку за все заплачено. В последних версиях Perl переменная $& требует меньших расходов, чем две остальные.

Модификаторы шаблонов

Мы обсудим отдельные операторы поиска по шаблону буквально через мгновение, но сначала хотелось бы сказать еще об одном, общем для них: *модификаторах*.

Сразу за последним ограничителем оператора m//, s///, qr// или tr/// можно поместить один или несколько необязательных модификаторов, в любом порядке. Для большей ясности модификаторы обычно пишутся как «модификатор /o» и читаются «модификатор слэш о», даже если конечный ограничитель отличен от слэша (косой черты). (Иногда говорят «флаг» или «опция», имея в виду «модификатор»; это тоже нормально.)

Некоторые модификаторы изменяют режим действия отдельного оператора, подробно мы опишем их позже. Другие изменяют то, как интерпретируется регулярное выражение, и мы поговорим о них сейчас. Операторы m//, s/// и qr//[1] – все принимают следующие модификаторы после своего конечного ограничителя:

Модификатор	Значение
/i	Игнорировать различия в регистре
/s	Разрешить . устанавливать соответствие переводу строки и игнорировать устаревшую переменную $*
/m	Разрешить ^ и $ устанавливать соответствие рядом со встроенным \n
/x	Игнорировать (почти все) пробельные символы и разрешить в шаблоне комментарии
/o	Компилировать шаблон только один раз

Модификатор /i указывает на необходимость поиска в верхнем и нижнем регистрах (и в заглавном регистре «title» в Unicode). Например, /perl/i найдет соответствие в строках "PROPERLY" и "Perlaceous" (среди прочих). Прагма use locale также может повлиять на критерий эквивалентнсти. (Ее влияние может оказаться пагубным, если строки содержат Unicode.)

Модификаторы /s и /m не делают ничего оригинального. Они влияют на способ, которым Perl осуществляет поиск в строке, содержащей символы перевода строки. Но они относятся не к определению наличия символа \n в анализируемой строке, а к тому, должен ли Perl *допускать*, что последовательность символов содержит одну строку (/s) или несколько (/m), т. к. некоторые метасимволы по-разному действуют в зависимости от того, предполагается ли их использование в режиме, ориентированном на строки, или нет.

[1] Оператор tr/// не принимает регулярные выражения, поэтому к нему неприменимы эти модификаторы.

Обычно метасимвол "." соответствует любому символу, *кроме* символа перевода строки, потому что традиционно он предназначается для поиска символов в *строчке* (*line*). Однако при наличии модификатора /s метасимвол "." может соответствовать и символу перевода строки, поскольку этот модификатор сообщает Perl о необходимости игнорировать то обстоятельство, что строка (string) может содержать символы перевода строки. (Модификатор /s также заставляет Perl игнорировать устаревшую переменную $*, которую, надеемся, вы тоже игнорируете.) Напротив, модификатор /m изменяет интерпретацию метасимволов ^ и $, позволяя им устанавливать соответствие рядом с символами перевода строки внутри нее, а не только по концам строки. Примеры см. в разделе «Позиции» далее в этой главе.

Модификатор /o управляет перекомпиляцией шаблона. Если только в качестве разделителей не выбраны одинарные кавычки (m'*PATTERN*', s'*PATTERN*'*REPLACEMENT*' или qr'*PATTERN*'), то все переменные в шаблоне будут интерполироваться (что может повлечь повторную компиляцию шаблона) при каждом вычислении оператора шаблона. Для того чтобы этот шаблон компилировался один и только один раз, используйте модификатор /o. Это позволяет избежать дорогостоящей рекомпиляции на этапе исполнения. Однако указание /o содержит в себе обещание, что мы не будем изменять переменные, входящие в шаблон. Если же их все-таки изменить, Perl не обратит на это никакого внимания. Лучшее управление рекомпиляцией достигается при использовании оператора кавычек qr//. Подробности см. далее в этой главе в разделе «Интерполяция переменных».

Модификатор /x является *экс*прессивным: он позволяет *экс*плуатировать пробельные символы и пояснительные комментарии в целях улучшения понятности вашего шаблона и даже *экс*пансии шаблона через границы символов перевода строки.

Это означает, что /x изменяет значение пробельных символов (и символа #): вместо того чтобы при поиске «притвориться» обычными символами, они превращаются в метасимволы, которые, как ни странно, ведут себя в результате так, как и должны пробельные символы (и символ комментария). Следовательно, благодаря /x пробелы, символы табуляции и перевода строки используются для форматирования, как в обычном коде Perl. Также благодаря этому модификатору за символом #, обычно не являющимся в шаблоне каким-либо особенным, можно размещать комментарии, продолжающиеся до конца текущей строчки (line) в строке (string) шаблона.[1] Если действительно требуется найти пробельный символ (или символ #), придется поместить его в класс символов либо поставить перед ним обратную косую черту, либо записать его в восьмеричном или шестнадцатеричном коде. (Но обычно поиск пробельных символов осуществляется с помощью последовательностей \s* или \s+, поэтому на практике такая ситуация возникает редко.)

---

[1] Следите за тем, чтобы в комментарий не попал ограничитель шаблона, т. к., следуя своему правилу «сначала найти конец», Perl не сможет определить, что вы не собирались закончить шаблон в этом месте.

Совместное применение этих функций значительно способствует превращению обычных регулярных выражений в доступный для чтения язык. В соответствии с духом TMTOWTDI теперь есть больше одного способа записи данного регулярного выражения. На самом деле их больше двух:

```
m/\w+:(\s+\w+)\s*\d+/;        # Слово, двоеточие, пробел, слово, пробел, цифры.

m/\w+: (\s+ \w+) \s* \d+/x;   # Слово, двоеточие, пробел, слово, пробел, цифры.

m{
    \w+:                      # Найти слово и двоеточие.
         \s+                  # Найти один или несколько пробелов.
         \w+                  # Найти еще одно слово.
    )                         # (конец группы)
    \s*                       # Найти ноль или несколько пробелов.
    \d+                       # Найти несколько цифр
}x;
```

Эти новые метасимволы мы объясним далее в этой главе. (Данный раздел должен был рассказывать о модификаторах шаблонов, но, впав в *экс*таз по поводу /x, мы не смогли удержаться в заданных рамках.) Вот регулярное выражение, отыскивающее повторяющиеся слова в абзацах и украденное прямо из «Perl. Библиотека программиста». В нем используются модификаторы /x и /i, а также модификатор /g, который будет описан ниже.

```
# Найти в абзацах повторяющиеся слова, возможно, пересекая границы строчек.
#   Применяется /x для пробелов и комментариев, /i для поиска обоих `is'
#   в "Is is this ok?", и /g для поиска всех дубликатов.
$/ = "";        # режим "paragrep"
while (<>) {
    while ( m{
                \b            # начать с границы слова
                (\w\S+)       # найти похожий на слово фрагмент,
                (
                    \s+       # отделенный каким-нибудь пробелом,
                    \1        # и еще раз этот же фрагмент.
                ) +           # Повторять, сколько возможно,
                \b            # до следующей границы слова
            }xig
        )
    {
        print "повтор слова '$1' в абзаце $.\n";
    }
}
```

Если применить этот код к данной главе, можно получить что-то вроде:

```
повтор слова 'THE' в абзаце 100
```

В данном случае мы знаем, что это повторение является преднамеренным.

Оператор m// (поиск)

```
EXPR =~ m/PATTERN/cgimosx
EXPR =~ /PATTERN/cgimosx
EXPR =~ ?PATTERN?cgimosx

m/PATTERN/cgimosx
/PATTERN/cgimosx
?PATTERN?cgimosx
```

Оператор m// ищет строку в скаляре *EXPR*, соответствующую шаблону *PATTERN*. Если ограничителем служит символ / или ?, то присутствие начальной m не обязательно. Символы ? и ' имеют специальное значение в случае применения в качестве ограничителя: первый служит для однократного поиска, второй подавляет интерполяцию переменных и шесть escape-последовательностей трансляции (\U и тому подобных, описываемых ниже).

Если значением *PATTERN* окажется нулевая строка (потому ли, что вы так задали его с помощью //, потому ли, что интерполируемая переменная дала значение в виде пустой строки), то для поиска используется последнее успешно выполненное регулярное выражение, не скрытое во внутреннем блоке (или в split, grep или map).

В скалярном контексте оператор возвращает в случае успешного выполнения значение «истина» (1), а в противном случае значение «ложь» ("").

В булевом контексте обычно можно видеть такую форму:

```
if ($shire =~ m/Baggins/) { ... }  # искать Baggins в $shire
if ($shire =~ /Baggins/)  { ... }  # искать Baggins в $shire

if ( m#Baggins# )         { ... }  # искать прямо в $_
if ( /Baggins/ )          { ... }  # искать прямо в $_
```

В списковом контексте m// возвращает список подстрок, соответствующих захватывающим скобкам в шаблоне (т. е. $1, $2, $3 и т. д.), как описывается далее, в разделе «Захват и кластеризация» данной главы. Нумерованным переменным по-прежнему присваиваются значения, даже если возвращается список. Если поиск в списковом контексте оказывается безуспешным, возвращается нулевой список. А если успешен, но скобок захвата в шаблоне нет (как и модификатора /g), возвращается список со значением (1). Поскольку при неудаче поиска возвращается нулевой список, этот формат m// можно применять в булевом контексте, но только в косвенном виде через присвоение списку:

```
if (($key,$value) = /(\w+): (.*)/) { ... }
```

Допустимые для m// (в любом его обличье) модификаторы показаны в табл. 5.1.

Таблица 5.1. Модификаторы m//

Модификатор	Значение
/i	Игнорировать различия в регистре
/m	Разрешить ^ и $ устанавливать соответствие рядом со встроенным \n
/s	Разрешить . устанавливать соответствие переводу строки и игнорировать устаревшую переменную $*
/x	Игнорировать (почти все) пробельные символы и разрешить в шаблоне комментарии
/o	Компилировать шаблон только один раз
/g	Глобальный поиск всех соответствий
/cg	Разрешить продолжение поиска после неудачи поиска /g

Первые пять модификаторов относятся к регулярному выражению и были описаны выше. Последние два меняют режим работы самого оператора поиска. Модификатор /g задает глобальный поиск, т. е. поиск всех соответствий шаблона в строке. Однако его действие зависит от контекста. В списковом контексте m//g возвращает список всех найденных соответствий. Найдем, например, все места, где упоминается "perl", "Perl", "PERL" и т. д.:

```
if (@perls = $paragraph =~ /perl/gi) {
    printf "Perl упомянут %d раз.\n", scalar @perls;
}
```

Если в шаблоне с /g нет захватывающих скобок, возвращаются полные соответствия. Если скобки есть, возвращаются только захваченные строки. Пусть имеется такая строка:

```
$string = "password=xyzzy verbose=9 score=0";
```

Пусть также требуется использовать ее для такой инициализации хеша:

```
%hash = (password => "xyzzy", verbose => 9, score => 0);
```

Кроме того, конечно, у вас есть не список, а строка. Чтобы получить соответствующий список, можно применить оператор m//g в списковом контексте и захватить все пары ключ/значение, имеющиеся в строке:

```
%hash = $string =~ /(\w+)=(\w+)/g;
```

Последовательность (\w+) захватывает слово из алфавитно-цифровых символов. См. раздел «Захват и кластеризация» далее в этой главе.

В скалярном контексте модификатор /g задает *поступательный поиск*, что заставляет Perl искать следующее соответствие для той же переменной, начиная с позиции, следующей сразу за последним найденным соответствием. Утверждение \G представляет эту позицию в строке (см. описание \G ниже в этой главе в разделе «Позиции»). Если в дополнение к /g указан модифи-

катор /c («continue»), то когда /g заканчивает работу, последний неудачный поиск не сбрасывает указатель позиции.

Если в качестве ограничителя выступает ?, как в ?*PATTERN*?, то поиск выполняется, как обычный /*PATTERN*/, но при этом мы ищем только одно соответствие между вызовами оператора reset. Такой прием может дать удобный способ оптимизации, если во время прогона программы требуется найти не все соответствия шаблону, а только первое. Оператор осуществляет поиск при каждом вызове, пока, наконец, что-нибудь не найдет, после чего выключается и будет возвращать «ложь», пока не будет явно включен снова с помощью reset. Работу по отслеживанию состояния поиска Perl берет на себя.

Оператор ?? полезнее всего тогда, когда обычный поиск по шаблону ищет последнее, а не первое вхождение:

```
open DICT, "/usr/dict/words" or die "Невозможно открыть words: $!\n";
while (<DICT>) {
    $first = $1 if ?(^neur.*)?;
    $last  = $1 if /(^neur.*)/;
}
print $first,"\n";          # выводит "neurad"
print $last,"\n";           # выводит "neurypnology"
```

Оператор reset сбрасывает только экземпляры ??, скомпилированные в том же пакете, что и вызов reset. Высказывание m?? эквивалентно высказыванию ??.

Оператор s/// (подстановка)

```
LVALUE =~ s/PATTERN/REPLACEMENT/egimosx
s/PATTERN/REPLACEMENT/egimosx
```

Этот оператор ищет в строке шаблон *PATTERN* и, если он найден, заменяет найденную подстроку текстом *REPLACEMENT*. (Модификаторы описываются ниже.)

```
$lotr = $hobbit;            # Просто копируем Хоббита
$lotr =~ s/Bilbo/Frodo/g;   #   и легко пишем продолжение.
```

Значение, возвращаемое оператором s/// (одинаковое в скалярном и списковом контекстах), равно количеству успешных выполнений (оно может быть больше одного при использовании модификатора /g, описанного выше). При неуспехе, поскольку количество замен равно нулю, он возвращает значение «ложь» (""), численный эквивалент 0.

```
if ($lotr =~ s/Bilbo/Frodo/) { print "Продолжение успешно написано." }
$change_count = $lotr =~ s/Bilbo/Frodo/g;
```

Часть оператора, определяющая заменяющий текст, рассматривается как строка в двойных кавычках. В строке замены можно использовать любые из описанных выше переменных шаблонов с динамической областью видимости ($`, $&, $', $1, $2 и т. д.) и любые другие связанные с двойными кавычками

приемы, которые вам заблагорассудится применить. Вот, например, как найти все строки "revision", "version", "release" и заменить каждую из них эквивалентом из заглавных букв с помощью управляющего символа \u в заменяющей части:

```
s/revision|version|release/\u$&/g;  # Используйте | для обозначения "или" в шаблоне
```

В контексте двойных кавычек разыменовываются все скалярные переменные, а не только эти необычные. Допустим, у нас есть хеш %Names, содержащий соответствие номеров версий внутренним названиям проектов; например, $Names{"3.0"} содержит кодовое имя "Isengard". Можно с помощью s/// найти номера версий и заменить их соответствующими названиями проектов:

```
s/version ([0-9.]+)/the $Names{$1} release/g;
```

В строке замены $1 возвращает то, что захватила первая (и единственная) пара скобок. (Можно также указать \1, как в шаблоне, но такое использование в замене считается предосудительным. В обычной строке, заключенной в двойные кавычки, \1 означает <Ctrl>+<A>.)

Если шаблон *PATTERN* представляет собой нулевую строку, то вместо него используется последнее успешно выполненное регулярное выражение. Интерполяция переменных происходит в обеих строках, *PATTERN* и *REPLACEMENT*, но *PATTERN* интерполируется всякий раз, когда оператор s/// выполняется в целом, а *REPLACEMENT* интерполируется каждый раз, когда шаблон найден. (*PATTERN* может находить несколько соответствий при одном выполнении, если указан модификатор /g.)

Как и выше, первые пять модификаторов в табл. 5.2 изменяют действие регулярного выражения; они те же, что в m// и qr//. Последние два изменяют сам оператор подстановки.

Таблица 5.2. Модификаторы s///

Модификатор	Значение
/i	Игнорировать различия в регистре (при поиске)
/m	Разрешить ^ и $ устанавливать соответствие рядом со встроенным \n
/s	Разрешить . устанавливать соответствие переводу строки и игнорировать устаревшую переменную $*
/x	Игнорировать (почти все) пробельные символы и разрешить в шаблоне комментарии
/o	Компилировать шаблон только один раз
/g	Глобальная замена, т. е. всех вхождений
/e	Вычислять правую часть как выражение

Модификатор /g используется с оператором /// для замены всех, а не только первого найденного, значений *PATTERN* значением *REPLACEMENT*. Оператор s///g действует как глобальный поиск и замена, производя сразу все изменения,

подобно списковому m//g, за исключением того, что m//g ничего не изменяет. (А s///g не является последовательным поиском, в отличие от скалярного m//g.)

Модификатор /e рассматривает *REPLACEMENT* как фрагмент кода Perl, а не как интерполируемую строку. Результат выполнения этого кода выступает в качестве заменяющей строки. Например, s/([0-9]+)/sprintf("%#x", $1)/ge преобразует все числа в шестнадцатеричные, заменяя, например, 2581 на 0xb23. Или, предположим, в нашем прежнем примере мы не уверены в наличии названий для всех версий, поэтому все остальные хотим оставить без изменений. Творчески применив форматирование /x, можно сказать:

```
s{
    version
    \s+
    (
        [0-9.]+
    )
}{
    $Names{$1}
        ? "the $Names{$1} release"
        : $&
}xge;
```

Правая часть s///e (или, в данной части, нижняя часть) проходит синтаксическую проверку и компилируется на этапе компиляции, как и остальная часть нашей программы. Синтаксические ошибки обнаруживаются во время компиляции, а исключительные ситуации времени исполнения не перехватываются. Каждый дополнительный /e после первого (например, /ee, /eee и т. д.) эквивалентен вызову eval *STRING* и его применению к результату выполнения кода, по одному для каждого лишнего /e. При этом вычисляется результат кодового выражения, а исключительные ситуации перехватываются в специальной переменной $@. Дополнительные сведения есть в разделе «Программные шаблоны» далее в этой главе.

Модификация строк en passant

Иногда требуется, чтобы новая, модифицированная строка не портила старую, на которой она основана. Вместо того чтобы писать:

```
$lotr = $hobbit;
$lotr =~ s/Bilbo/Frodo/g;
```

можно объединить эти команды в одну. Порядок старшинства операторов требует, чтобы присвоение было заключено в круглые скобки, поскольку в большинстве случаев они применяют =~ к выражению.

```
($lotr = $hobbit) =~ s/Bilbo/Frodo/g;
```

В отсутствие скобок была бы изменена строка $hobbit, а в $lotr было бы записано количество сделанных замен, каковой результат не представил бы особого интереса.

Применять оператор s/// непосредственно к массиву нельзя. Для этого нужен цикл. По счастливому совпадению создание псевдонимов в for/foreach в сочетании с использованием $_ в качестве переменной цикла по умолчанию лежит в основе стандартной идиомы Perl для поиска и замены в каждом элементе массива:

```
for (@chapters) { s/Bilbo/Frodo/g }  # Сделать замену в каждой главе
s/Bilbo/Frodo/g for @chapters;       # То же самое.
```

Как и для простых скалярных переменных, можно сочетать подстановку с присвоением, если требуется также сохранить исходные значения:

```
@oldhues = ('bluebird', 'bluegrass',  'bluefish', 'the blues');
for (@newhues = @oldhues) { s/blue/red/ }
print "@newhues\n";           # выводит: redbird redgrass redfish the reds
```

Идиоматическим способом выполнения повторных подстановок в той же самой переменной является применение цикла, выполняющегося один раз. Вот пример канонизации пробельных символов в переменной:

```
for ($string) {
    s/^\s+//;       # отбросить ведущие пробельные символы
    s/\s+$//;       # отбросить замыкающие пробельные символы
    s/\s+/ /g;      # сжать внутренние пробельные символы
}
```

что дает такой же результат, как и:

```
$string = join(" ", split " ", $string);
```

Можно также оранизовать такой же цикл с присваиванием, какой рассматривался в случае массива:

```
for ($newshow = $oldshow) {
    s/Fred/Homer/g;
    s/Wilma/Marge/g;
    s/Pebbles/Lisa/g;
    s/Dino/Bart/g;
}
```

Когда глобальная подстановка оказывается недостаточно глобальной

Иногда модификатора /g оказывается недостаточно для того, чтобы выполнить все необходимые изменения, т. к. или подстановка должна происходить справа налево, или нужно, чтобы длина $` изменялась между поисками. Обычно можно достичь желаемого, повторно вызывая s///. Однако нам нужно, чтобы цикл остановился, когда s/// окончательно окажется неуспешным, поэтому он должен быть помещен в условный оператор, и в основной части цикла не остается ничего для выполнения. Поэтому мы просто пишем 1, что довольно скучно, но иногда скука – это лучшее, на что можно рас-

считывать. Вот несколько примеров использования этих странных зверюшек из регулярных выражений, продолжающих возникать тут и там:

```
# расставить в нужных местах целого числа запятые
1 while s/(\d)(\d\d\d)(?!\d)/$1,$2/;

# заменить табуляцию на пробелы по 8 в колонке
1 while s/\t+/` ` x (length($&)*8 - length($`)%8)/e;

# удалить (вложенные (даже глубоко вложенные (как это))) замечания
1 while s/\([^()]*\)//g;

# удалить слова, повторяющиеся дважды (и трижды (и четырежды...))
1 while s/\b(\w+) \1\b/$1/gi;
```

В последнем случае требуется цикл, иначе:

```
Paris in THE THE THE THE spring.
```

будет превращено в:

```
Paris in THE THE spring.
```

что может вызвать перед глазами того, кто немного знает французский, образ Париса, сидящего в артезианском колодце с ледяным чаем, т. к. «the» по-французски – «чай».[1] Парижанина, конечно, не обманешь.

Оператор tr/// (транслитерации)

```
LVALUE =~ tr/SEARCHLIST/REPLACEMENTLIST/cds
tr/SEARCHLIST/REPLACEMENTLIST/cds
```

Для приверженцев *sed* предоставляется оператор `y///` в качестве синонима `tr///`. По этой причине нельзя вызывать функции с именем `y`, как и функции с именами `q` или `m`. Во всех остальных отношениях `y///` совпадает с `tr///`, и больше мы об этом говорить не будем.

Этот оператор может показаться не вписывающимся в главу о поиске по шаблону, поскольку он не использует шаблоны. Он просматривает строку символ за символом и заменяет каждый символ, оказывающийся в *SEARCHLIST* (это не регулярное выражение), на соответствующий символ из *REPLACEMENTLIST* (это не строка замены). Однако он несколько похож на `m//` и `s///`, и в нем даже можно применять операторы привязки `=~` и `!~`, поэтому мы описываем его здесь. (Операторы `qr//` и `split` выполняют поиск по шаблону, но с ними не используются операторы привязки, поэтому они находятся в другом месте книги. Куда мы вас и отсылаем.)

Транслитерация возвращает число замененных или удаленных символов. Если строка не задана с помощью операторов `=~` или `!~`, то изменения прово-

[1] Spring – весна; родник; Paris – Париж; Парис, сын Приама. – *Примеч. перев.*

дятся в строке $_. В *SEARCHLIST* и *REPLACEMENTLIST* могут определяться диапазоны последовательных символов с помощью дефиса:

```
$message =~ tr/A-Za-z/N-ZA-Mn-za-m/;    # rot13 шифрование.
```

Заметьте, что диапазон вроде A-Z предполагает линейный набор символов типа ASCII. Но в каждом наборе символов есть свои представления о том, как они упорядочены и, следовательно, о том, какие символы попадают в конкретный диапазон. Разумный принцип состоит в использовании только таких диапазонов, начало и конец которых находятся в алфавите одного регистра (a-e, A-E) или являются цифрами (0-4). Ко всем остальным нужно относиться с подозрительностью. Если есть сомнения, приведите набор символов полностью: ABCDE.

В *SEARCHLIST* и *REPLACEMENTLIST* интерполяция переменных, как в строках в двойных кавычках, не производится; можно, однако, использовать последовательности с обратной косой чертой, соответствующие конкретным символам, например \n или \015.

В табл. 5.3 перечислены модификаторы, применимые к оператору tr///. Они совершенно отличны от тех, которые применяются к m//, s/// или qr//, даже если выглядят одинаково.

Таблица 5.3. Модификаторы tr///

Модификатор	Значение
/c	Дополнение к *SEARCHLIST*
/d	Удалить найденные, но не замененные символы
/s	Подавить повторяющиеся замененные символы

Если задан модификатор /c, осуществляется дополнение набора символов в *SEARCHLIST*; т. е. реальный список для поиска состоит из всех символов, *отсутствующих* в *SEARCHLIST*. В случае применения Unicode таких символов может оказаться много, но поскольку они хранятся логически, а не физически, можно не беспокоиться о том, что не хватит памяти.

Модификатор /d превращает tr/// в оператор «замены-удаления»: любые символы, указанные в *SEARCHLIST*, для которых не задана замена в *REPLACEMENTLIST*, удаляются. (Это дает несколько большую гибкость, чем в некоторых программах *tr*(1), удаляющих все, что они находят в *SEARCHLIST*.)

Если задан модификатор /s, то последовательности символов, превращаемых в один и тот же символ, сжимаются до одного этого символа.

При использовании модификатора /d список *REPLACEMENTLIST* всегда интерпретируется точно так, как он задан. Без него, когда *REPLACEMENTLIST* короче *SEARCHLIST*, последний символ копируется до достижения нужной длины. Если *REPLACEMENTLIST* представляет собой нулевую строку, в него копируется *SEARCHLIST*, что оказывается полезным, если требуется просто подсчитать символы, не заменяя их. Это полезно также для сжатия символов с ключом /s.

```
tr/aeiou/!/;                  # заменить все гласные на!
tr{/\\\r\n\b\f. }{_};         # заменить необычные символы на подчеркивание

tr/A-Z/a-z/ for @ARGV;        # перевести в нижний регистр ASCII

$count = ($para =~ tr/\n//); # подсчитать количество переводов строки в $para
$count = tr/0-9//;            # подсчитать цифры в $_

$word =~ tr/a-zA-Z//s;        # bookkeeper -> bokeper

tr/@$%*//d;                   # удалить указанные
tr#A-Za-z0-9+/##cd;           # удалить символы не из base64

# изменить en passant
($HOST = $host) =~ tr/a-z/A-Z/;

$pathname =~ tr/a-zA-Z/_/cs; # заменить небуквы (ASCII) на подчеркивание

tr [\200-\377]
   [\000-\177];               # сбросить 8-й разряд
```

Если в *SEARCHLIST* один и тот же символ повторяется несколько раз, то используется только первый, поэтому команда

```
tr/AAA/XYZ/
```

поменяет все символы A на X (в `$_`).

Хотя переменные не интерполируются в `tr///`, того же эффекта можно добиться с помощью `eval` *EXPR*:

```
$count = eval "tr/$oldlist/$newlist/";
die if $@;  # распространяет исключительную ситуацию при недопустимом содержании
              eval
```

Еще одно замечание: не используйте `tr///` для перевода текста в верхний или нижний регистр. В этом случае следует предпочесть последовательности `\U` или `\L` в строках, заключенных в двойные кавычки (или эквивалентные функции `uc` и `lc`), поскольку они учитывают национальные установки и Unicode, а `tr/a-z/A-Z/` не учитывает. Кроме того, в строках Unicode последовательность `\u` и соответствующая ей функция `ucfirst` учитывают понятие «заглавного регистра» (titlecase), что в некоторых языках отличается от простого преобразования в верхний регистр.

Метасимволы и метазнаки

Теперь, полюбовавшись на все замысловатые клетки, можем вернуться назад и посмотреть на тварей, которые в этих клетках находятся, на те забавно выглядящие значки, которые помещают внутрь шаблонов. Сейчас вы должны уже привыкнуть к тому обстоятельству, что эти значки не являются обычным кодом Perl, таким как вызовы функций или арифметические операторы. Регулярные выражения представляют собой маленький самостоятельный язык, удобно устроившийся в Perl. (В каждом из нас есть немного джунглей.)

При всей своей мощи и выразительности шаблоны в Perl распознают те же 12 обычных метасимволов, которые имеются во многих других пакетах, поддерживающих регулярные выражения:

```
\ | ( ) [ { ^ $ * + ? .
```

Некоторые из них ломают правила, делая особенными располагающиеся за ними символы, которые при других обстоятельствах ничем особенным не отличаются. Мы не хотим называть длинные последовательности символами, поэтому, когда такие последовательности образуются, мы называем их *метазнаками* (metasymbols, или иногда просто symbols). Но на верхнем уровне эти двенадцать метасимволов представляют собой все, о чем вы (и Perl) должны думать. Все остальное проистекает из них.

Некоторые простые метасимволы (`.` `^` `$`) действуют самостоятельно. Они не оказывают непосредственного влияния на соседние символы. Другие метасимволы действуют как префиксные операторы, управляя тем, что следует за ними, например `\`. Третьи действуют как постфиксные операторы, управляя тем, что непосредственно предшествует им, например, `*`, `+` и `?`. Один метасимвол, `|`, действует как инфиксный оператор, располагаясь между управляемыми им операндами. Есть даже структурирующие метасимволы, действующие как охватывающие операторы, управляющие тем, что в них содержится, такие как `(...)` и `[...]`. Особенно важны круглые скобки, поскольку они задают границы для `|` с внутренней стороны и для `*`, `+`, и `?` с внешней.

Если запоминать только один из этих двенадцати метасимволов, нужно выбрать обратную косую черту. (Ну... еще круглые скобки.) Это связано с тем, что обратная косая черта отключает все остальные. Когда обратная косая черта предшествует в шаблоне символу, не являющемуся буквой или цифрой, она превращает этот символ в литерал. Если необходимо буквально найти по шаблону один из этих двенадцати метасимволов, то его нужно написать с обратной косой чертой впереди. Поэтому `\.` соответствует действительной точке, `\$` – действительному знаку доллара, `\\` – действительной обратной косой черте и т. д. Это называют «обходом» метасимвола («escaping», «quoting» или «backslashing»). (Конечно, вы уже знаете, что обратная косая черта используется для подавления интерполяции переменных в строках, заключенных в двойные кавычки.)

Обратная косая черта превращает метасимвол в литеральный, но действие ее на последующий буквенно-цифровой символ оказывается иным. Она берет то, что было обычным, и превращает его в особенное. Это значит, что вместе они образуют метазнак. Алфавитный список этих метазнаков можно найти далее в табл. 5.7.

Таблицы метазнаков

В последующих таблицах в колонке «Атомарный» указывается «да», если этот метазнак поддается количественному определению (может соответство-

вать чему-то, обладающему в какой-то мере шириной). Мы также применяли многоточие (...) для представления «чего-то еще». Пожалуйста, посмотрите следующее далее описание, чтобы выяснить, что означает ..., если это непонятно из однострочного объяснения в таблице.

В табл. 5.4 приведены основные обычные метазнаки. Первые четыре из них являются структурирующими, и о них говорилось выше, тогда как последние три – это просто метасимволы. Примером атома служит метасимвол . , поскольку он соответствует чему-то, имеющему ширину (в данном случае ширину символа); ^ и $ представляют собой примеры утверждений, поскольку соответствуют чему-то, имеющему нулевую ширину, и вычисляются, только чтобы посмотреть, истинны они или нет.

Таблица 5.4. Общие метасимволы регулярных выражений

Символ	Атомарный	Значение
\...	По-разному	Сделать обычным следующий не буквенно-цифровой метасимвол и сделать метасимволом (может быть) следующий буквенно-цифровой символ
...|...	Нет	Чередование (соответствие одному или другому)
(...)	Да	Группировка (рассматривать воедино)
[...]	Да	Класс символов (соответствие одному символу из набора)
^	Нет	Истинно в начале строки (или, возможно, после любого символа перевода строки)
.	Да	Соответствие одному символу (обычно, кроме символа перевода строки)
$	Нет	Истинно в конце строки (или, возможно, перед любым символом перевода строки)

Квантификаторы, которым ниже посвящен отдельный раздел, показывают, сколько раз предшествующий атом (т. е. отдельный символ или их группа) должен быть найден. Они перечислены в табл. 5.5.

Таблица 5.5. Квантификаторы регулярных выражений

Квантификатор	Атомарный	Значение
*	Нет	0 или более совпадений (максимальный)
+	Нет	1 или более совпадений (максимальный)
?	Нет	1 или 0 или более совпадений (максимальный)
{*COUNT*}	Нет	Ровно *COUNT* совпадений
{*MIN*,}	Нет	Не меньше *MIN* совпадений (максимальный)
{*MIN*, *MAX*}	Нет	Не меньше *MIN* совпадений, но не больше *MAX* (максимальный)
*?	Нет	0 или более совпадений (минимально)

Таблица 5.5 (продолжение)

Квантификатор	Атомарный	Значение
+?	Нет	1 или более совпадений (минимальный)
??	Нет	0 или 1 совпадений (минимальный)
{*MIN*,}?	Нет	Не меньше *MIN* совпадений (минимальный)
{*MIN*, *MAX*}?	Нет	Не меньше *MIN*, но не больше *MAX* совпадений (минимальный)

Минимальный квантификатор пытается найти как можно *меньше* символов в отпущенных ему пределах, максимальный – как можно *больше*. Например, .+ обеспечивает соответствие по крайней мере одному символу в строке, но будет соответствовать всем им, если дать возможность. Эти возможности описываются далее в разделе «Маленькая машинка, которая /(не) может?/» данной главы.

Заметьте, что квантификаторы нельзя квантифицировать.

Мы хотим обеспечить открытый синтаксис для новых типов метазнаков. Учитывая, что мы могли работать всего лишь с дюжиной метасимволов, мы выбрали для использования в произвольных синтаксических расширениях ранее недопустимую для регулярных выражений последовательность. Все эти метазнаки имеют вид (?*KEY*...); т. е. скобка (парная), за которой следует вопросительный знак, за которым следует *KEY* и оставшаяся часть подшаблона. Символ *KEY* указывает на конкретное расширение синтаксиса регулярных выражений. Полный их список приведен в табл. 5.6. Большинство из них является структурирующими, поскольку основываются на скобках, но у них есть и дополнительное значение. Квантифицироваться могут только атомы, поскольку они представляют нечто действительно существующее (потенциально).

Таблица 5.6. Расширенные последовательности регулярных выражений

Расширение	Атомарный	Значение
(?#...)	Нет	Отбросить комментарий
(?:...)	Да	Скобки только для кластера, без захвата
(?imsx-imsx)	Нет	Включить/отключить модификаторы шаблона
(?imsx-imsx:...)	Да	Скобки только для кластера плюс модификаторы
(?=...)	Нет	True, если выполняется утверждение после данной точки
(?!...)	Нет	True, если не выполняется утверждение после данной точки
(?<=...)	Нет	True, если выполняется утверждение перед данной точкой
(?<!...)	Нет	True, если не выполняется утверждение перед данной точкой
(?>...)	Да	Совпадение с подшаблоном без ссылки на найденный текст

Расширение	Атомарный	Значение
`(?{...})`	Нет	Выполнить встроенный код Perl
`(??{...})`	Да	Поиск регулярного выражения из встроенного кода Perl
`(?(...)...|...)`	Да	Поиск по шаблону if-then-else
`(?(...)...)`	Да	Поиск по шаблону if-then

И наконец, в табл. 5.7 перечислены все ваши любимые буквенно-цифровые метазнаки. (Знаки, обрабатываемые на проходе интерполяции переменных, помечены в колонке «атомарный» символом тире «–», поскольку машина регулярных выражений их даже не увидит.)

Таблица 5.7. Буквенно-цифровые метазнаки регулярных выражений

Символ	Атомарный	Значение
`\0`	Да	Совпадение с символом null (ASCII NUL)
`\NNN`	Да	Совпадение с символом, заданным в восьмеричной системе, до `\377`
`\n`	Да	Совпадение с *n*-й ранее захваченной строкой (десятичный)
`\a`	Да	Совпадение с символом звонка (BEL)
`\A`	Нет	True в начале строки
`\b`	Да	Совпадение с символом возврата (BS)
`\b`	Нет	True на границе слова
`\B`	Нет	True не на границе слова
`\cX`	Да	Совпадение с управляющим символом Control-*X* (`\cZ`, `\c[` и т. д.)
`\C`	Да	Совпадение с одним байтом (C `char`) даже в utf8 (опасно)
`\d`	Да	Совпадение с любой цифрой
`\D`	Да	Совпадение с любым символом – не цифрой
`\e`	Да	Совпадение с символом escape (ASCII ESC, не обратная косая черта)
`\E`	—	Конец трансляции регистра (`\L`, `\U`) или метакавычки (`\Q`)
`\f`	Да	Совпадение с символом подачи страницы (FF)
`\G`	Нет	True в позиции конца совпадения или перед `m//g`
`\l`	—	Нижний регистр только для следующего символа
`\L`	—	Нижний регистр до `\E`
`\n`	Да	Совпадение с символом перевода строки (обычно NL, но CR на Mac'ах)
`\N{NAME}`	Да	Совпадение с именованным символом (`\N{greek:Sigma}`)
`\p{PROP}`	Да	Совпадение с любым символом с поименованным свойством
`\P{PROP}`	Да	Совпадение с любым символом без поименованного свойства

Таблица 5.7 (продолжение)

Символ	Атомарный	Значение
\Q	—	Цитировать (де-мета) метасимволы до \E
\r	Да	Совпадение с символом возврата каретки (обычно CR, но NL на Mac'ах)
\s	Да	Совпадение с любым пробельным символом
\S	Да	Совпадение с любым непробельным символом
\t	Да	Совпадение с символом табуляции (HT)
\u	—	Заглавный регистр только для следующего символа
\U	—	Верхний регистр (не заглавный регистр) до \E
\w	Да	Совпадение с любым «символом слова» (буквенно-цифровым или «_»)
\W	Да	Совпадение с любым «несимволом слова»
\x{*abcd*}	Да	Совпадение с символом, заданным в шестнадцатеричной системе
\X	Да	Совпадение со строкой Unicode «комбинированная символьная последовательность»
\z	Нет	True только в конце строки
\Z	Нет	True в конце строки или перед символом перевода строки

Если в \p или \P имя свойства состоит из одного символа, фигурные скобки не обязательны. Если в \x шестнадцатеричное число состоит из двух и менее цифр, фигурные скобки не обязательны. Фигурные скобки всегда обязательны в \N.

В классах символов (квадратных скобках) могут использоваться только метасимволы, в описании которых присутствует «Совпадение с ... символом» или «Совпадение с любым». Это значит, что классы символов могут содержать только метазнаки, описывающие другие конкретные наборы символов или описывающие отдельные конкретные символы. Конечно, эти метасимволы можно использовать вне классов символов наряду с остальными неклассифицирующими метазнаками. Обратите внимание, что \b – это два совершенно разных «зверя»: внутри класса символов это символ забоя (backspace), но утверждение границы слова вне его.

Существует некое перекрытие между символами, которые может находить шаблон, и символами, которые может интерполировать обычная строка в двойных кавычках. Поскольку регулярные выражения обрабатываются за два прохода, иногда возникает неоднозначность в том, на котором из проходов должен обрабатываться данный символ. Если возникает неоднозначность, то на проходе, осуществляющем интерполяцию переменных, интерпретация таких символов откладывается для анализатора регулярных выражений.

Однако на проходе интерполяции переменных можно отложить интерпретацию до анализатора регулярных выражений, только если известно, что про-

водится анализ регулярного выражения. Можно задавать регулярные выражения как обычные строки в двойных кавычках, но тогда нужно следовать правилам, применяемым к обычным строкам в двойных кавычках. Все предыдущие метазнаки, соответствующие фактическим символам, будут работать, даже если они не откладываются до анализатора регулярных выражений. Но нельзя использовать другие метасимволы в обычных двойных кавычках (или аналогичных конструкциях типа `...`, qq(...), qx(...) или эквивалентных встроенных документах (here documents). Для того чтобы строка анализировалась как регулярное выражение без выполнения поиска, следует использовать оператор qr// (заключить в кавычки регулярное выражение).

Заметьте, что escape-последовательности трансляции регистра и метакавычек (\U и компания) должны обрабатываться на этапе интерполяции переменных, поскольку назначение этих метазнаков состоит в воздействии на способ интерполяции переменных. Если подавить интерполяцию переменных с помощью одинарных кавычек, то не удастся и получить escape-последовательности трансляции. Ни переменные, ни escape-последовательности трансляции (\U и т. д.) не разыменовываются ни в строках в одинарных кавычках, ни в операторах одинарных кавычек m'...' или qr'...'. Даже при выполнении интерполяции эти escape-последовательности трансляции игнорируются, если они появляются как результат интерполяции переменных, поскольку в этом случае уже слишком поздно влиять на интерполяцию переменных.

Хотя оператор транслитерации не принимает регулярные выражения, любой метасимвол из обсуждавшихся нами, который соответствует конкретному отдельному символу, работает и в операции tr///. Остальные – нет (за исключением обратной косой черты, которая как всегда продолжает работать обратным способом).

Специальные символы

Как уже говорилось, в шаблоне все, что не является специальным, соответствует самому себе. Это значит, что /a/ ищет «a», /=/ ищет «=» и т. д. Однако некоторые символы не так просто ввести с клавиатуры, а если вам это и удастся, то они не будут видны в выводе; этим славятся управляющие символы. В регулярном выражении Perl распознает следующие псевдонимы символов в двойных кавычках:

Escape-последовательность	Значение
\0	Символ Null (ASCII NUL)
\a	Звонок (BEL)
\e	Escape (ESC)
\f	Подача страницы (FF)
\n	Перевод строки (NL, CR на Mac)
\r	Возврат каретки (CR, NL на Mac)
\t	Табуляция (HT)

Как и в строках, заключенных в двойные кавычки, Perl отличает в шаблонах следующие четыре метазнака:

`\cX`

Именованный управляющий символ, например, `\cC` для <Ctrl>+<C>, `\cZ` для <Ctrl>+<Z>, `\c[` для <Esc> и `\c?` для <Del>.

`\NNN`

Символ, заданный с помощью двух- или трехзначного восьмеричного кода. Ведущий `0` не обязателен, за исключением чисел, меньших `010` (8 десятичное), поскольку (в отличие от строк в двойных кавычках) код из одной цифры всегда считается ссылкой на подстроку текста, найденного по шаблону. Несколько цифр интерпретируются как *n*-ая ссылка, если ранее вы захватили по шаблону не менее *n* подстрок (где *n* считается десятичным числом); в противном случае они интерпретируются как символ в восьмеричной системе.

`\x{LONGHEX}`

`\xHEX`

Номер символа, заданный в виде одной или двух шестнадцатеричных цифр (`[0-9a-fA-F]`), например `\x1B`. Формат из одной цифры можно использовать, только если следующий символ не является шестнадцатеричной цифрой. Если применяются фигурные скобки, цифр может быть сколько угодно, в результате чего можно получить символ Unicode. Например, `\x{262f}` соответствует символу Unicode «Инь – Ян».

`\N{NAME}`

Именованный символ, например `\N{GREEK SMALL LETTER EPSILON}`, `\N{greek:epsilon}` или `\N{epsilon}`. Для этого требуется прагма `use charnames`, описанная в главе 31 «Модули прагм», которая также определяет те разновидности имен, которые можно использовать (`":long"`, `":full"`, `":short"` соответственно, связанные с тремя показанными только что стилями).

Список всех имен символов Unicode можно найти в ближайшем документе со стандартами Unicode или в *ПУТЬ_К_PERLLIB/unicode/Names.txt*.

Групповые метазнаки

Три особых метазнака выступают в качестве групповых, каждому из которых соответствует «любой» символ (для некоторых значений слова «любой»). Ими являются точка (`".."`), `\C` и `\X`. Ни один из них нельзя использовать в классах символов. В них не допускается налие точки, потому что она будет соответствовать (почти) любому существующему символу, являясь своего рода всеобщим классом символов. Если необходимо все исключить или все включить, нет особого смысла в использовании класса символов. Особые групповые символы `\C` и `\X` имеют особый структурирующий смысл, который не очень хорошо согласуется с идеей выбора одного символа Unicode, что является тем уровнем, на котором работают классы символов.

Метасимвол «точка» соответствует любому отдельному символу, кроме символа перевода строки. (А с модификатором /s соответствует и ему тоже.) Как и для любого из двенадцати метасимволов в шаблоне, для буквального поиска точки ее нужно преобразовать с помощью обратной косой черты. Например, следующая команда проверяет, заканчивается ли имя файла точкой и расширением из одного символа:

```
if ($pathname =~ /\.(.)\z/s) {
    print "Оканчивается на $1\n";
}
```

Первая точка, преобразованная обратной косой чертой, представляет собой литеральный символ, а вторая говорит «найти любой символ». Последовательность \z указывает, что соответствие следует искать только в конце строки, а модификатор /s разрешает, чтобы точка соответствовала и символу перевода строки. (Да, использование символа перевода строки в качестве расширения файла выглядит не очень красиво, но это не значит, что так не бывает.)

Символ точки чаще всего используется с квантификатором. Выражение .* соответствует максимальному числу символов, тогда как .*? соответствует минимальному числу символов. Но иногда она применяется без квантификатора для ширины: /(..):(..):(..)/ ищет три поля, каждое из которых имеет размер в два символа и отделяется двоеточием.

Если точка присутствует в шаблоне, компилированном с прагмой use utf8, находящейся в лексической области видимости, то она будет соответствовать любому символу Unicode. (Для этого не предполагается необходимость use utf8, но в противном случае возникнут неприятности. К тому моменту, когда вы будете это читать, использование этой прагмы, возможно, не потребуется.)

```
use utf8;
use charnames qw/:full/;
$BWV[887] = "G\N{MUSIC SHARP SIGN} minor";
($note, $black, $mode) = $BWV[887] =~ /^([A-G])(.)\s+(\S+)/;
print "That's lookin' sharp!\n" if $black eq chr(9839);
```

Метасимвол \X ищет символ в более широком смысле. Он, в действительности, ищет строку из одного или нескольких символов Unicode, называемую «combining character sequence» – комбинированной символьной последовательностью. Такая последовательность состоит из базового символа, за которым следуют символы «пометок» (диакритические знаки, такие как седиль « ☐ » и умляут « ¨ » (dieresis)), объединяющиеся с этим базовым символом в одну логическую единицу. \X в точности эквивалентен (?:\PM\pM*). Это позволяет находить один логический символ, даже если в действительности он состоит из нескольких отдельных символов. Длина соответствия /\X/ больше одного символа, если найдена комбинация из нескольких символов. (Это длина в символах, а не в байтах.)

Если вы работаете с Unicode и действительно хотите получить один байт, а не один символ, можете воспользоваться метазнаком `\C`. Он всегда соответствует одному байту (именно одному типу `char` языка C), даже если при этом возникает расхождение с потоком символов Unicode. См. соответствующие предупреждения в главе 15.

Классы символов

При поиске по шаблону можно искать любой символ, который обладает (или не обладает) некоторым свойством. Есть четыре способа задания классов символов. Можно задавать классы символов традиционным способом – при помощи квадратных скобок и перечисляя возможные символы или прибегнув к любому из трех мнемонических сокращений: традиционные классы Perl, новые для Perl свойства Unicode или стандартные классы POSIX. Каждое из этих сокращений ищет только один символ из своего набора. Снабдите их квантификаторами для поиска более крупных промежутков, например, `\d+` ищет одну или несколько цифр. (Естественная ошибка – считать, что `\w` соответствует слову. Для поиска слова используйте `\w+`.)

Пользовательские классы символов

Перечисление символов, заключенное в квадратные скобки, называется *классом символов* и соответствует любому одному из символов этого списка. Например, `[aeiouy]` ищет букву, являющуюся гласной в английском языке. (Для уэльсского добавьте `"w"`, для шотландского – `"r"`.) Для поиска правой квадратной скобки поставьте перед ней обратную косую черту или поместите первой в списке.

Диапазоны символов могут задаваться с помощью дефиса и обозначения типа `a-z`. Можно объединять несколько диапазонов, например, `[0-9a-fA-F]` соответствует одной шестнадцатеричной «цифре». Можно использовать обратную косую черту, чтобы защитить дефис, который иначе будет интерпретироваться как ограничитель диапазона, либо поместить дефис в начале или конце класса (прием, который наверняка менее читаем, но более традиционен).

Символ вставки ^ (или циркумфлекс, или «крышка», или «стрелка вверх») перед классом символов инвертирует класс, в результате чего он соответствует любому символу, *не* входящему в список. (Чтобы найти символ вставки, либо *не* ставьте его первым, либо, еще лучше, поставьте перед ним обратную косую черту.) Например, `[^aeiouy]` соответствует любому символу, который не является гласной. Однако будьте осторожны с отрицанием классов символов, поскольку вселенная символов расширяется. Например, этот класс символов соответствует согласным, а также пробелам, символам перевода строки и чему угодно (включая гласные) в кириллице, греческом и почти любом другом алфавите, не говоря уже обо всех китайских, японских и корейских иероглифах. А в один прекрасный день, может быть, даже в ал-

фавитах Cirth, Tengwar и Klingon[1]. (И уж наверняка в критском письме B и этрусском.) Поэтому лучше, может быть, задать согласные явно, например, как `[cbdfghjklmnpqrstvwxyz]` или, для краткости, `[b-df-hj-np-tv-z]`. (Это также решает проблему с латинской буквой «y», которая должна быть одновременно в обоих местах, что не получится при использовании дополнения класса.)

Внутри класса символов можно применять обычные символьные метазнаки, (см. раздел «Специальные символы» этой главы), такие как `\n`, `\t`, `\c`*X*, `\`*NNN* и `\N{`*NAME*`}`. Кроме того, в классе символов можно использовать `\b`, означающий возврат, так же как в строках, заключенных в двойные кавычки. Обычно при поиске по шаблону он обозначает границу слова. Но утверждения нулевой ширины не имеют смысла в классах символов, поэтому в них `\b` возвращается к своему обычному значению в строках. Можно также использовать любые предопределенные классы символов из описываемых далее в этой главе (классические, Unicode или POSIX), но не пытайтесь применять их в качестве границ диапазона: это бессмысленно, поэтому символ `-` будет интерпретирован буквально.

Все остальные метазнаки утрачивают свое особое значение, находясь в квадратных скобках. В частности, не допускается применение трех общих групповых символов: `.`, `\X` и `\C`. Первое часто вызывает удивление, но не имеет смысла использовать универсальный класс символов внутри ограниченного, зато часто нужно найти точку в составе класса символов, например при поиске имен файлов. Бессмысленно также задавать квантификаторы, утверждения или чередование внутри класса символов, поскольку символы интерпретируются индивидуально. Например, `[fee|fie|foe|foo]` означает то же самое, что и `[feio|]`.

Классические сокращенные обозначения классов в Perl

С самого начала в Perl имелся ряд сокращенных обозначений классов символов. Они перечислены в табл. 5.8. Все они являются буквенными метазнаками с косой чертой впереди, и в каждом случае версия на верхнем регистре представляет собой отрицание версии на нижнем регистре. Значение их не является вполне фиксированным, как можно было бы ожидать: на него могут оказывать влияние национальные установки. Даже если вы не работаете с национальными установками, значение может изменяться при выходе нового стандарта Unicode, вводящего дополнительные буквы и цифры. (Чтобы сохранить старые значения байтов, можно использовать прагму `use bytes`.

[1] Cirth (Кирт, Кертар) и Tengwar (Тенгвар) – изобретенные Дж. Р.Р. Толкиеном два вида эльфийских алфавитов – рунический (Кертар) и буквенный (Тенгвар). Klingon (Клингон) – алфавит синтетического языка «клингонов» – злобных персонажей игры Star Trek. Создан лингвистом Марком Окрэндом (Mark Okrand) для большей достоверности киноверсии этой игры (от компании Paramount). – *Примеч. ред.*

Значения utf8 поясняются в разделе «Свойства Unicode» далее в этой главе. В любом случае значения utf8 являются надмножеством значений байтов.)

Таблица 5.8. Классические классы символов

Символ	Значение	Как байт	Как utf8
\d	Цифра	[0-9]	\p{IsDigit}
\D	Не цифра	[^0-9]	\P{IsDigit}
\s	Пробельный символ	[\t\n\r\f]	\p{IsSpace}
\S	Непробельный символ	[^ \t\n\r\f]	\P{IsSpace}
\w	Символ слова	[a-zA-Z0-9_]	\p{IsWord}
\W	Не (символ слова)	[^a-zA-Z0-9_]	\P{IsWord}

(Мы знаем, что большинство слов не содержит цифры и символы подчеркивания; \w ищет «слова» в смысле типичного языка программирования. Или в смысле Perl, если на то пошло.)

Эти метазнаки могут использоваться снаружи или внутри квадратных скобок как самостоятельно, так и в составе создаваемого класса символов:

```
if ($var =~ /\D/)        { warn "содержит нецифру" }
if ($var =~ /[^\w\s.]/)  { warn "содержит не-(слово, пробел, точка)" }
```

Свойства Unicode

К свойствам Unicode обращаются с помощью \p{*PROP*} и его дополнительного множества \P{*PROP*}. Для тех редких свойств, имена которых состоят из одного символа, фигурные скобки не обязательны, как в \pN для обозначения числового символа (не обязательно десятичного, римские числа тоже являются числовыми символами). Эти классы свойств можно использовать самостоятельно или объединять в создаваемом классе символов:

```
if ($var =~ /^\p{IsAlpha}+$/)      { print "одни буквы" }
if ($var =~ s/[\p{Zl}\p{Zp}]/\n/g) { print "желающие исправлены на перевод строки" }
```

Некоторые свойства непосредственно определены в стандарте Unicode, другие являются составными, определяемыми Perl на основе стандартных свойств. Zl и Zp представляют собой стандартные свойства Unicode, это разделители строк и параграфов, а IsAlpha определяется в Perl как класс свойств, объединяющий стандартные свойства Ll, Lu, Lt и Lo (т. е. буквы в нижнем и верхнем регистрах, в заголовочном регистре или другом). В версии Perl 5.6.0 требуется задать use utf8, чтобы эти свойства работали. Это ограничение в будущем будет ослаблено.

Свойств очень много. Мы перечислим те, о которых нам известно, но список неизбежно будет неполным. Вероятно, в новых версиях Unicode появятся новые свойства, и можно даже определять собственные. Подробнее об этом далее.

Консорциум Unicode (Unicode Consortium) производит электронные ресурсы, которые обращаются в различные файлы, используемые Perl в его реализации Unicode. Подробнее об этих файлах сказано в главе 15. Хороший обзор Unicode содержится в документе *ПУТЬ_К_PERLLIB/unicode/Unicode3.html*, где *ПУТЬ_К_PERLLIB* может быть выведен с помощью команды:

```
perl -MConfig -le 'print $Config{privlib}'
```

Большинство свойств Unicode имеют вид `\p{IsPROP}`. Буквы `Is` сделаны необязательными, поскольку они очень часто употребляются, но для улучшения читаемости их можно оставить.

Свойства Unicode в Perl

Во-первых, в табл. 5.9 перечислены составные свойства Perl. Они определены так, чтобы быть достаточно близкими к стандартным определениям POSIX классов символов.

Таблица 5.9. Составные свойства Unicode

Свойство	Эквивалент
IsASCII	[\x00-\x7f]
IsAlnum	[\p{IsLl}\p{IsLu}\p{IsLt}\p{IsLo}\p{IsNd}]
IsAlpha	[\p{IsLl}\p{IsLu}\p{IsLt}\p{IsLo}]
IsCntrl	\p{IsC}
IsDigit	\p{Nd}
IsGraph	[^\pC\p{IsSpace}]
IsLower	\p{IsLl}
IsPrint	\P{IsC}
IsPunct	\p{IsP}
IsSpace	[\t\n\f\r\p{IsZ}]
IsUpper	[\p{IsLu}\p{IsLt}]
IsWord	[_\p{IsLl}\p{IsLu}\p{IsLt}\p{IsLo}\p{IsNd}]
IsXDigit	[0-9a-fA-F]

Perl предоставляет также следующие составные свойства для каждой из основных категорий стандартных свойств Unicode (см. следующий раздел):

Свойство	Значение	Нормативность
IsC	Необычные управляющие коды и т. п.	Да
IsL	Буквы	Частично
IsM	Пометки	Да
IsN	Числа	Да
IsP	Пунктуация	Нет
IsS	Символы	Нет
IsZ	Разделители (сепараторы? зепараторы?)	Да

Стандартные свойства Unicode

В табл. 5.10 перечислены основные стандартные свойства Unicode, производные от всех категорий символов. Всякий символ принадлежит только одной категории. Некоторые свойства являются нормативными, другие – чисто информативными. В Стандарте Unicode можно найти стандартные разглагольствования о том, насколько нормативна нормативная информация и насколько информативна информативная информация.

Таблица 5.10. Стандартные свойства Unicode

Свойство	Значение	Normative
IsCc	Прочие, управляющие	Да
IsCf	Прочие, форматирование	Да
IsCn	Прочие, не назначены	Да
IsCo	Прочие, частное использование	Да
IsCs	Прочие, суррогат	Да
IsLl	Буква, нижний регистр	Да
IsLm	Буква, модификатор	Нет
IsLo	Буква, прочие	Нет
IsLt	Буква, заглавный регистр	Да
IsLu	Буква, верхний регистр	Да
IsMc	Специальный знак, объединение	Да
IsMe	Специальный знак, заключение	Да
IsMn	Знак, диакритический	Да
IsNd	Число, десятичная цифра	Да
IsNl	Число, буквенное	Да
IsNo	Число, прочие	Да
IsPc	Пунктуация, соединительный символ	Нет
IsPd	Пунктуация, черточка	Нет
IsPe	Пунктуация, закрытие	Нет
IsPf	Пунктуация, закрывающая кавычка	Нет
IsPi	Пунктуация, открывающая кавычка	Нет
IsPo	Пунктуация, прочие	Нет
IsPs	Пунктуация, открытие	Нет
IsSc	Символ, валюты	Нет
IsSk	Символ, модификатор	Нет
IsSm	Символ, математический	Нет
IsSo	Символ, прочие	Нет
IsZl	Разделитель, строк	Да
IsZp	Разделитель, абзацев	Да
IsZs	Разделитель, пробел	Да

Другой полезный набор свойств касается того, может ли данный символ быть подвергнут декомпозиции (канонической или совместимой) и разбит на более простые символы. При канонической декомпозиции никакая информация о форматировании не теряется. Совместимая декомпозиция может утратить информацию о форматировании, такую, например, как сведения о том, является ли символ надстрочным индексом.

Свойство	Утраченная информация
IsDecoCanon	Нет
IsDecoCompat	Кое-какая (что-то из нижеследующего)
IsDCcircle	Кружок вокруг символа
IsDCfinal	Предпочтение последней позиции (арабский)
IsDCfont	Предпочтение варьирующегося шрифта
IsDCfraction	Характеристика простой дроби
IsDCinitial	Предпочтение начальной позиции (арабский)
IsDCisolated	Предпочтение изолированной позиции (арабский)
IsDCmedial	Предпочтение средней позиции (арабский)
IsDCnarrow	Характеристика «узкий» (narrow)
IsDCnoBreak	Предпочтение «неразрывный» (пробел или тире)
IsDCsmall	Характеристика «малый»
IsDCsquare	Символы ККЯ (Корея, Китай, Япония), обведенные рамкой
IsDCsub	Нижний индекс
IsDCsuper	Верхний индекс
IsDCvertical	Поворот (с горизонтального в вертикальное)
IsDCwide	Характериситка «широкий» (wide)
IsDCcompat	Особенность (identity), разное

Вот некоторые свойства, интересные тем, кто осуществляет двунаправленный вывод:

Свойство	Значение
IsBidiL	Слева направо (арабский, еврейский)
IsBidiLRE	Слева направо, встраивание
IsBidiLRO	Слева направо, переопределение
IsBidiR	Справа налево
IsBidiAL	Справа налево, арабский
IsBidiRLE	Справа налево, встраивание
IsBidiRLO	Справа налево, переопределение

(продолжение)

Свойство	Значение
IsBidiPDF	Прочитать формат направленности (pop directional format)
IsBidiEN	Число в европейском формате
IsBidiES	Разделитель числа в европейском формате
IsBidiET	Терминатор числа в европейском формате
IsBidiAN	Арабское число
IsBidiCS	Разделитель числа
IsBidiNSM	Метка непробельного символа (nonspacing mark)
IsBidiBN	Граничный символ (boundary neutral)
IsBidiB	Разделитель абзацев
IsBidiS	Разделитель сегментов
IsBidiWS	Пробельный символ
IsBidiON	Other Neutrals
IsMirrored	Обратить при использовании справа налево

Следующие свойства классифицируют различные слоговые азбуки в соответствии с гласными звуками:

```
IsSylA      IsSylE      IsSylO      IsSylWAA    IsSylWII
IsSylAA     IsSylEE     IsSylOO     IsSylWC     IsSylWO
IsSylAAI    IsSylI      IsSylU      IsSylWE     IsSylWOO
IsSylAI     IsSylII     IsSylV      IsSylWEE    IsSylWU
IsSylC      IsSylN      IsSylWA     IsSylWI     IsSylWV
```

Например, `\p{IsSylA}` соответствует `\N{KATAKANA LETTER KA}`, но не `\N{KATAKANA LETTER KU}`.

Теперь, когда мы в основном рассмотрели свойства Unicode 3.0, следует указать, что несколько наиболее таинственных из них не реализованы в версии Perl 5.6.0, поскольку отчасти его реализация основывалась на Unicode 2.0 и такие вещи, как двунаправленный алгоритм, все еще разрабатывались. Однако к моменту чтения данной книги отсутствующие свойства вполне могут быть реализованы, поэтому мы на всякий случай их перечислили.

Свойства блоков Unicode

Некоторые свойства Unicode имеют вид `\p{In`*SCRIPT*`}`. (Обратите внимание на разницу между `Is` и `In`.) Свойства `In` предназначены для проверки блоковых диапазонов конкретного шрифта *SCRIPT*. Если у вас есть символ и вас интересует, принадлежит ли он греческому алфавиту, можно проверить это с помощью

```
print "It's Greek to me!\n" if chr(931) =~ /\p{InGreek}/;
```

При этом проверяется, находится ли символ «в» допустимом для данного шрифта диапазоне. Отрицание можно задать с помощью `\P{In`*SCRIPT*`}`, чтобы установить, что нечто не находится в конкретном блоке шрифта; например, `\P{InDingbats}` проверяет, содержит ли строка не-«как его там». В число свойств блоков входят следующие:

```
InArabic       InCyrillic     InHangulJamo   InMalayalam    InSyriac
InArmenian     InDevanagari   InHebrew       InMongolian    InTamil
InArrows       InDingbats     InHiragana     InMyanmar      InTelugu
InBasicLatin   InEthiopic     InKanbun       InOgham        InThaana
InBengali      InGeorgian     InKannada      InOriya        InThai
InBopomofo     InGreek        InKatakana     InRunic        InTibetan
InBoxDrawing   InGujarati     InKhmer        InSinhala      InYiRadicals
InCherokee     InGurmukhi     InLao          InSpecials     InYiSyllables
```

Не говоря уже о таких зубодробительных:

```
InAlphabeticPresentationForms          InHalfwidthandFullwidthForms
InArabicPresentationForms-A            InHangulCompatibilityJamo
InArabicPresentationForms-B            InHangulSyllables
InBlockElements                        InHighPrivateUseSurrogates
InBopomofoExtended                     InHighSurrogates
InBraillePatterns                      InIdeographicDescriptionCharacters
InCJKCompatibility                     InIPAExtensions
InCJKCompatibilityForms                InKangxiRadicals
InCJKCompatibilityIdeographs           InLatin-1Supplement
InCJKRadicalsSupplement                InLatinExtended-A
InCJKSymbolsandPunctuation             InLatinExtended-B
InCJKUnifiedIdeographs                 InLatinExtendedAdditional
InCJKUnifiedIdeographsExtensionA       InLetterlikeSymbols
InCombiningDiacriticalMarks            InLowSurrogates
InCombiningHalfMarks                   InMathematicalOperators
InCombiningMarksforSymbols             InMiscellaneousSymbols
InControlPictures                      InMiscellaneousTechnical
InCurrencySymbols                      InNumberForms
InEnclosedAlphanumerics                InOpticalCharacterRecognition
InEnclosedCJKLettersandMonths          InPrivateUse
InGeneralPunctuation                   InSuperscriptsandSubscripts
InGeometricShapes                      InSmallFormVariants
InGreekExtended                        InSpacingModifierLetters
```

А вот победитель:

```
InUnifiedCanadianAboriginalSyllabics
```

Чтобы получить более новый список этих свойств блоков символов, обратитесь к файлу *ПУТЬ_К_PERLLIB/unicode/In/*.pl*. Заметьте, что эти свойства `In` только проверяют, входит ли символ в блок символов, выделенный данному шрифту. Нет гарантии, что в этом диапазоне определены все символы, поэтому нужно также проверить одно из свойств `Is`, обсуждавшихся выше, чтобы узнать, определен ли данный символ. Нет также гарантии, что неко-

торый язык не использует символы вне выделенного ему блока. В частности, во многих европейских языках смешаны символы расширенной латиницы и Latin-1.

Но долой уныние; если вам нужно какое-то свойство, которое отсутствует, то невелика проблема. Читайте дальше.

Задание собственных свойств символов

Для того чтобы определить собственное свойство, следует написать подпрограмму, именем которой будет название свойства (см. главу 6 «Подпрограммы»). Процедура должна быть определена в пакете, которому требуется это свойство (см. главу 10 «Пакеты»), а это значит, что если требуется использовать его в нескольких пакетах, то понадобится либо импортировать его из модуля (см. главу 11 «Модули»), либо наследовать его как метод класса из пакета, в котором он определен (см. главу 12 «Объекты»).

Если все это улажено, то подпрограмма должна возвратить данные в таком же формате, как файлы в каталоге *ПУТЬ_К_PERLLIB/unicode/Is*. Это значит, что должен возвращаться список символов или диапазонов символов в шестнадцатеричном виде, по одному в строчке (line). Если это диапазон, то два числа разделяются символом табуляции. Предположим, что нам нужно свойство, принимающее значение «истина», если символ находится в какой-либо из японских азбук, известных как хирагана и катакана (вместе их называют кана). Можно просто задать два диапазона, например:

```
sub InKana {
    return <<'END';
3040    309F
30A0    30FF
END
}
```

Либо можно определить свойство, используя названия существующих свойств:

```
sub InKana {
    return <<'END';
+utf8::InHiragana
+utf8::InKatakana
END
}
```

Можно также вычитать массивы символов с помощью префикса «-». Предположим, что нам нужны только фактические символы, а не диапазоны блоков символов. Отмести все неопределенные символы можно так:

```
sub IsKana {
    return <<'END';
+utf8::InHiragana
+utf8::InKatakana
```

```
-utf8::IsCn
END
}
```

Можно также начать с дополнительного набора символов, используя префикс «!»:

```
sub IsNotKana {
    return <<'END';
!utf8::InHiragana
-utf8::InKatakana
+utf8::IsCn
END
}
```

Сам Perl использует точно такие же приемы для определения своих «классических» классов символов (типа `\w`), когда мы включаем их в собственные классы символов (типа `[-.\w\s]`). Может показаться, что чем сложнее становятся правила, тем медленнее они выполняются, но на практике, после того как Perl рассчитает битовую матрицу для некоторого 64-разрядного образчика нашего свойства, он его кэширует, и больше пересчитывать матрицу не приходится. (Он делает это в 64-разрядных образчиках (swatches), поэтому для поиска ему даже не приходится декодировать наш utf8.) В результате все классы символов, встроенные или пользовательские, работают примерно с одинаковой скоростью, после того как они созданы.

Классы символов в стиле POSIX

В отличие от других сокращенных обозначений классов символов в Perl, обозначения классов символов в стиле POSIX, `[:CLASS:]`, могут использоваться *только* при создании других классов символов, то есть внутри другой пары квадратных скобок. Например `/[.,[:alpha:][:digit:]]/` осуществляет поиск одного символа, который является литеральной точкой (поскольку она находится в классе символов) или запятой, или буквой алфавита, или цифрой.

Классы POSIX, имеющиеся в Perl версии 5.6, показаны в табл. 5.11.

Таблица 5.11. Классы символов POSIX

Класс	Значение
`alnum`	Любой буквенно-цифровой символ, т. е. `alpha` или `digit`
`alpha`	Любая буква. (Это значительно больше букв, чем вы думаете, если только вы не думаете на Unicode, но и тогда это достаточно много.)
`ascii`	Любой символ с порядковым значением от 0 до 127
`cntrl`	Любой управляющий символ. Обычно это символы, которые сами не выводятся, но некоторым образом управляют терминалом, например, перевод строки, подача страницы, возврат – все они являются управляющими символами. Символы с порядковым номером до 32 чаще всего классифицируются как управляющие

Таблица 5.11 (продолжение)

Класс	Значение
digit	Символ, представляющий десятичную цифру, например, от 0 до 9. (В Unicode включает и другие символы.) Эквивалентен \d
graph	Любой буквенно-цифровой символ или знак пунктуации
lower	Буква на нижнем регистре
print	Любой буквенно-цифровой символ или знак пунктуации, или пробел
punct	Любой знак пунктуации
space	Любой пробельный символ. Включает символы табуляции, перевода строки, подачи страницы и возврата каретки (и много чего еще в Unicode). Эквивалентен \s
upper	Любая буква в верхнем (или заглавном) регистре
word	Любой символ идентификатора, буквенно-цифровой или подчеркивания
xdigit	Любая шестнадцатеричная цифра. Хотя это может показаться глупым (достаточно [0-9a-fA-F]), включен для полноты

Можно создавать отрицание классов символов POSIX, помещая перед именем класса ^ вслед за [:. (Это расширение Perl.) Например:

POSIX	Классический
[:^digit:]	\D
[:^space:]	\S
[:^word:]	\W

Если прагма use utf8 не запрашивается, но запрашивается прагма use locale, то классы прямо коррелируют с эквивалентными функциями в интерфейсе библиотеки C *isalpha*(3) (кроме word, являющегося расширением Perl, копирующим \w).

Если используется прагма utf8, то классы символов POSIX в точности эквивалентны соответствующим свойствам Is, которые перечислены в табл. 5.9. Например, эквивалентны [:lower:] и \p{Lower}, за исключением того, что классы POSIX могут применяться только при создании классов символов, тогда как для свойств Unicode такого ограничения нет, и они могут использоваться в тех же местах шаблонов, где и сокращенные обозначения классов Perl \s и \w.

Квадратные скобки являются частью конструкции в стиле POSIX [::], а не частью класса символов в целом. Поэтому можно писать такие шаблоны, как /^[[:lower:][:digit:]]+$/ для поиска строки, состоящей только из букв на нижнем регистре или цифр (и, возможно, замыкающего символа перевода строки). В частности, следующая команда не работает:

```
42 =~ /^[:digit:]$/         # НЕВЕРНО
```

Это неверно, потому что класс POSIX не находится внутри класса символов. Но это класс символов, представляющий символы «:», «i», «t», «g» и «d». Perl безразлично, что символ «:» задан дважды.

Вот что нужно было написать в действительности:

```
42 =~ /^[[:digit:]]+$/
```

Классы символов POSIX [.cc.] и [=cc=] распознаются, но порождают ошибку, в которой сообщается, что они не поддерживаются. Попытка применения любого класса символов POSIX в более старых версиях Perl провалится, причем, возможно, без слов. Если вы собираетесь использовать классы символов POSIX, лучше всего затребовать новую версию Perl, сказав:

```
use 5.6.0;
```

Квантификаторы

Если не указано иное, то для каждого элемента в регулярном выражении отыскивается только одно совпадение. Для такого шаблона, как /nop/, должны быть найдены все эти символы, один за другим. Такие слова, как «panoply» и «xenophobia», подходят, потому что не имеет значения, *где* находится совпадение.

Если нам надо, чтобы подходили как «xenophobia», так и «Snoopy», то шаблон /nop/ использовать нельзя, поскольку он требует наличия только одного «o» между «n» и «p», а в «Snoopy» их два. В таких случаях на помощь приходят *квантификаторы*: они сообщают, сколько раз что-то должно совпадать, вместо однократного совпадения по умолчанию. Квантификаторы в регулярном выражении – как циклы в программе. В действительности, если представить себе регулярное выражение как программу, то они и *есть* циклы. Некоторые циклы точные, типа «повторить это совпадение только пять раз» ({5}). Другие задают нижнюю и верхнюю границы счетчика совпадений, типа «повторить это совпадение не меньше двух, но не больше четырех раз» ({2,4}). Кроме того, вообще нет установленной верхней границы, вроде «совпадение по крайней мере дважды, но столько раз, сколько угодно» ({2,}).

В табл. 5.12 показаны квантификаторы, которые Perl распознает в шаблоне.

Таблица 5.12. Квантификаторы в регулярных выражениях

Максимальный	Минимальный	Допустимый диапазон
{*MIN*, *MAX*}	{*MIN*, *MAX*}?	Совпадение не меньше *MIN* раз, но не больше *MAX* раз
{*MIN*,}	{*MIN*,}?	Совпадение не меньше *MIN* раз
{*COUNT*}	{*COUNT*}?	Совпадение ровно *COUNT* раз
*	*?	0 или более раз (то же, что {0,})
+	+?	1 или более раз (то же, что {1,})
?	??	0 или 1 раз (то же, что {0,1})

Нечто, обозначаемое `*` или `?`, фактически не должно совпадать. Дело в том, что совпадение может произойти 0 раз и считаться успехом. Квантификатор `+` может быть более подходящим, поскольку предполагает хотя бы одно соответствие.

Пусть вас не смущает использование в предыдущей таблице слова «ровно». Оно относится только к счетчику повторений, а не к строке в целом. Например, `$n =~ /\d{3}/` не означает: «Равна ли длина строки ровно трем цифрам?» Здесь спрашивается, есть ли в `$n` место, в котором подряд находятся три цифры. Строка «101 Morris Street» удовлетворяет этому, как и строки «95472» и «1-800-555-1212». Все они содержат три цифры в одной или нескольких точках, и это все, о чем мы спрашивали. Для закрепления см. раздел «Позиции», где описывается использование утверждений позиций (как в `/^\d{3}$/`).

Имея возможность найти совпадение разное количество раз, максимальные квантификаторы выбирают максимальное число повторений. Поэтому, когда мы говорим: «Столько раз, сколько угодно», жадный квантификатор интерпретирует это как «столько раз, сколько сможешь унести», ограничиваясь лишь требованием, чтобы это не помешало поиску совпадений согласно последующим спецификациям. Если в шаблоне есть два квантификатора с открытыми концами, то, очевидно, оба они не могут «съесть» строку целиком: символы, которые использует одна часть совпадения, становятся недоступными части, следующей за ней. Все квантификаторы проявляют жадность, обделяя следующие за ними, при этом шаблон читается слева направо.

Это обычное поведение квантификаторов в регулярных выражениях. Однако Perl позволяет нам изменить поведение своих квантификаторов: поместив `?` после квантификатора, мы превратим его из максимального в минимальный. Это не значит, что минимальный квантификатор всегда будет давать совпадение из минимального числа повторений, разрешенных в его диапазоне, как не означает и того, что максимальный квантификатор всегда будет находить максимальное число совпадений, допустимое в его диапазоне. Успешным должно быть совпадение в целом, и минимальное совпадение возьмет столько, сколько нужно, чтобы быть успешным, и не более. (Минимальные квантификаторы удовлетворение ценят выше жадности.)

Например, при поиске:

```
"exasperate" =~ /e(.*)e/    #  $1 содержит теперь "xasperat"
```

`.*` соответствует `"xasperat"`, самой длинной из возможных строк. (Это значение также записывается в `$1`, как описывается в разделе «Захват и кластеризация» далее в этой главе.) Хотя имелось более короткое соответствие, при жадном поиске это не важно. Если в одной и той же точке есть возможность выбора, всегда возвращается более длинный вариант.

Сравните это со следующим:

```
"exasperate" =~ /e(.*?)e/   #  $1 содержит теперь "xasp"
```

Здесь применяется минимальная версия поиска, `.*?`. Добавление `?` к `*` заставляет `*?` вести себя противоположным образом: теперь, если в одной точке есть выбор из двух возможностей, всегда возвращается более *короткая* из них.

Хотя можно прочесть `*?` как «искать ноль или более чего-то, но предпочтительно ноль», это не значит, что всегда будет отыскиваться ноль символов. Если бы это было так, и в нашем примере `$1` была установлена в `""`, то не была бы найдена вторая `"e"`, т. к. она не следует за первой непосредственно.

Вы можете поинтересоваться, почему при минимальном поиске `/e(.*?)e/` Perl не вставил в `$1` строку `"rat"` . Ведь `"rat"` тоже находится между двумя `"e"` и короче `"xasp"`. В Perl выбор между максимальным и минимальным происходит только при наличии альтернатив соответствия, причем все они имеют одну и ту же начальную точку. Если есть два возможных соответствия, но начинаются они на разном смещении от начала строки, то их длина не играет роли, как и то, какой квантификатор используется – минимальный или максимальный. Из нескольких возможных соответствий побеждает всегда то, которое встретилось раньше. Лишь если несколько возможных совпадений начинаются в одной и той же точке, для принятия решения используется минимальное или максимальное соответствие. А если начальные точки различаются, то и решать нечего. Обычно Perl ищет *самое левое наибольшее* совпадение; при минимальном поиске оно становится *самым левым наименьшим*. Часть «самое левое» всегда постоянна и является доминирующим критерием.[1]

Есть два способа преодолеть «левый уклон» поиска по шаблону. Во-первых, можно использовать сначала жадный квантификатор (обычно `.*`), чтобы постараться «проглотить» начальные части строки. При поиске соответствия для жадного квантификатора сначала отыскивается самое длинное соответствие, в результате чего оставшаяся часть строки просматривается справа налево:

```
"exasperate" =~ /.*e(.*?)e/   #  $1 теперь содержит "rat"
```

Но будьте осторожны, поскольку общий результат поиска теперь содержит всю строку вплоть до этой точки.

Вторым способом преодоления стремления влево является использование утверждений позиций, обсуждаемых в следующем разделе.

---

[1] Не все механизмы регулярных выражений действуют таким образом. Некоторые привержены общей жадности, при которой всегда побеждает самое длинное совпадение, даже если оно находится дальше. Perl не таков. Можно сказать, что желание сильнее жадности (или расчетливости). Более формальное обсуждение этого и многих других принципов можно найти в разделе «Маленькая машинка, которая /(не) может?/».

Позиции

Некоторые конструкции в регулярных выражениях символизируют позиции в строке, которые должны быть найдены, представляющие собой место слева или справа от реального символа. Эти метазнаки являются примерами утверждений нулевой ширины, поскольку они не соответствуют фактическим символам строки. Часто мы называем их просто утверждениями (assertions). (Они называются также точками привязки (anchors), поскольку привязывают какую-то часть шаблона к определенной позиции.)

Есть возможность манипулирования позициями в строке без использования шаблонов. Встроенная функция `substr` позволяет извлекать подстроки и осуществлять присвоение подстрокам, измеряемым с начала строки, с конца строки или с конкретного числового смещения. Этого может оказаться достаточно при работе, например, с записями фиксированной длины. Шаблоны необходимы, только если числового смещения недостаточно. Но чаще всего смещений недостаточно или, по крайней мере, они недостаточно удобны в сравнении с шаблонами.

Начало: утверждения \ A и ^

Утверждение `\A` соответствует только началу строки, какой бы она ни была. Однако традиционным утверждением начала строки (string) и начала строчки (line) является `^`. Поэтому, если в шаблоне используется модификатор `/m`[1] и внутри строки есть символы перевода строки, `^` соответствует также любому месту в строке сразу за символом перевода строки:

```
/\Abar/      # Соответствует "bar" и "barstool"
/^bar/       # Соответствует "bar" и "barstool"
/^bar/m      # Соответствует "bar" и "barstool" and "sand\nbar"
```

В сочетании с модификатором `/g` модификатор `/m` позволяет находить соответствие `^` многократно в одной и той же строке:

```
s/^\s+//gm;              # Обрезать ведущие пробельные символы во всех строчках
$total++ while /^./mg;   # Подсчитать непустые строчки
```

Конец: утверждения \z, \Z и $

Метасимвол `\z` соответствует концу строки независимо от ее содержания. `\Z` соответствует позиции перед символом перевода строки в конце строки, если этот символ там есть, и в конце строки, если его нет. Метасимвол `$` обычно имеет тот же смысл, что и `\Z`. Однако если задан модификатор `/m` и внутри строки есть символы перевода строки, то `$` может соответствовать любому месту в строке перед символом перевода строки:

---

[1] Или вы установили устаревшую переменную `$*` равной `1` и не переопределяете `$*` с помощью модификатора `/s`.

```
/bot\z/      # Соответствует "robot"
/bot\Z/      # Соответствует "robot" и "abbot\n"
/bot$/       # Соответствует "robot" и "abbot\n"
/bot$/m      # Соответствует "robot" и "abbot\n" и "robot\nrules"

/^robot$/    # Соответствует "robot" и "robot\n"
/^robot$/m   # Соответствует "robot" и "robot\n" и "this\nrobot\n"
/\Arobot\Z/  # Соответствует "robot" и "robot\n"
/\Arobot\z/  # Соответствует только "robot" - - но почему вы не использовали eq?
```

Как и для ^, модификатор /m позволяет находить соответствие $ многократно в одной и той же строке, если задан модификатор /g. (В следующих примерах предполагается, что в $_ считана запись из нескольких строк; возможно, нужно установить перед чтением $/ равной "".)

```
s/\s*$//gm;   # Обрезать замыкающие пробелы во всех строках абзаца

while (/^([^:]+):\s*(.*)/gm ) {  # получить заголовок почтового сообщения
    $headers{$1} = $2;
}
```

В разделе «Интерполяция переменных» далее в этой главе мы расскажем о том, как можно интерполировать переменные в шаблонах: если $foo имеет значение «bc», то /a$foo/ эквивалентно /abc/. Здесь $ не соответствует концу строки. Чтобы $ соответствовал концу строки, он должен быть в конце шаблона, либо сразу за ним должна следовать вертикальная черта или закрывающая скобка.

Границы: утверждения \b и \B

Утверждение \b соответствует любой границе слова, определяемой как позиция между символом \w и символом \W, в любом порядке. Порядок \W\w соответствует границе начала слова, а порядок \w\W – границе конца слова. (Концы строки рассматриваются здесь как символы \W.) Утверждение \B соответствует любой позиции, которая *не* является границей слова, т. е. находится в середине \w\w или \W\W.

```
/\bis\b/   # Соответствует "what it is" и "that is it"
/\Bis\B/   # Соответствует "thistle" и "artist"
/\bis\B/   # Соответствует "istanbul" и "so- -isn't that butter?"
/\Bis\b/   # Соответствует "confutatis" и "metropolis near you"
```

Поскольку \W включает все знаки пунктуации (кроме подчеркивания), границы \b присутствуют в середине таких строк, как «isn't», «booktech@oreilly.com», «M.I.T.» и «key/value».

Внутри класса символов ([\b]) знак \b представляет символ возврата, а не границу слова.

Последовательный поиск

При использовании с модификатором /g функция pos позволяет читать или устанавливать смещение, по которому начнется очередной последовательный поиск:

```
$burglar = "Bilbo Baggins";
while ($burglar =~ /b/gi) {
    printf "Найдена B в позиции %d\n", pos($burglar)-1;
}
```

(Мы вычитаем из позиции единицу, поскольку мы ищем длину строки, а pos всегда указывает на позицию после найденного соответствия.)

Вышеприведенный код выводит:

```
Найдена B в позиции 0
Найдена B в позиции 3
Найдена B в позиции 6
```

После неудачного поиска позиция для поиска обычно сбрасывается в начало. Если дополнительно указать модификатор /c (от «continue»), то когда очередной поиск с модификатором /g окажется неудачным, указатель позиции не будет сброшен в начало. Это позволяет продолжить поиск, не возобновляя его с самого начала.

```
$burglar = "Bilbo Baggins";
while ($burglar =~ /b/gci) {        # ADD /c
    printf " Найдена B в позиции %d\n", pos($burglar)-1;
}
while ($burglar =~ /i/gi) {
    printf " Найдена I в позиции %d\n", pos($burglar)-1;
}
```

Кроме трех B, найденных ранее, Perl сообщает теперь об i, найденной в позиции 10. Без /c второй поиск в цикле начался бы с начала и нашел бы еще одну i в позиции 1.

Где вы остановились: утверждение \G

При мысли об использовании функции pos возникает соблазн разделать строку с помощью функции substr, но такой подход редко бывает правильным. Чаще, если уж мы начали с поиска по шаблону, то лучше так и продолжить. Однако если надо найти утверждение позиции, то, вероятно, мы ищем \G.

Утверждение \G представляет в шаблоне ту же точку, которую pos представляет вне его. Если мы осуществляем последовательный поиск в строке с модификатором /g (или воспользовались функцией pos для непосредственного вызова начальной точки), то можем использовать \G для задания позиции сразу за предыдущим найденным соответствием. То есть он соответствует месту непосредственно перед тем символом, текущая позиция кото-

рого будет определена посредством pos. Это позволяет запомнить, где мы остановились:

```
($recipe = <<'DISH') =~ s/^\s+//gm;
    Preheat oven to 451 deg. fahrenheit.
    Mix 1 ml. dilithium with 3 oz. NaCl and
    stir in 4 anchovies.  Glaze with 1 g.
    mercury.  Heat for 4 hours and let cool
    for 3 seconds.  Serves 10 aliens.
DISH

$recipe =~ /\d+ /g;
$recipe =~ /\G(\w+)/;           # $1 теперь "deg"
$recipe =~ /\d+ /g;
$recipe =~ /\G(\w+)/;           # $1 теперь "ml"
$recipe =~ /\d+ /g;
$recipe =~ /\G(\w+)/;           # $1 теперь "oz"
```

Метазнак \G часто применяется в цикле, как показано в следующем примере. Мы «останавливаемся» после каждой цифровой последовательности и проверяем в этом месте, нет ли за ней аббревиатуры. Если есть, то захватываем два следующих слова, в противном случае только одно следующее слово:

```
pos($recipe) = 0;                  # На всякий случай сбросим \G в 0
$recipe =~ /(\d+) /g ) {
    my $amount = $1;
    if ($recipe =~ / \G (\w{0,3}) \. \s+ (\w+) /x) {  # сокращ. + слово
        print "$amount $1 of $2\n";
    } else {
        $recipe =~ / \G (\w+) /x;                     # просто слово
        print "$amount $1\n";
    }
}
```

В результате получается:

```
451 deg of fahrenheit
1 ml of dilithium
3 oz of NaCl
4 anchovies
1 g of mercury
4 hours
3 seconds
10 aliens
```

Захват и кластеризация

Шаблоны позволяют группировать их части в подшаблоны и запоминать строки, соответствующие этим подшаблонам. Первое мы называем *кластеризацией* (*clustering*), а второе – *захватом* (*capturing*).

Захват

Чтобы захватить строку и в дальнейшем ее использовать, заключите в круглые скобки соответствующий ей подшаблон. Первая пара скобок хранит свою подстроку в переменной $1, вторая пара – в $2 и т. д. Скобок может быть сколько угодно; Perl просто продолжает определять новые нумерованные переменные, представляющие эти захваченные строки.

Некоторые примеры:

```
/(\d)(\d)/  # Найти две цифры, захватив их в $1 и $2
/(\d+)/     # Найти одну или более цифр, захватив их все в $1
/(\d)+/     # Найти цифру один или более раз, захватив последнюю в $1
```

Обратите внимание на различия между вторым и третьим шаблонами. Обычно требуется то, что делает вторая форма. Третья форма *не* создает несколько переменных для нескольких цифр. Скобки нумеруются тогда, когда шаблон компилируется, а не тогда, когда для него найдено соответствие.

Захваченные строки часто называют *ссылками на найденный текст* (*back-references*), поскольку они ссылаются на части захваченного текста. Есть два способа получить эти ссылки. Нумерованные переменные, которые мы видели, показывают, как получить ссылки на найденный текст, находясь вне шаблона, но в самом шаблоне они не работают. Для этого приходится использовать \1, \2 и т. д.[1] Поэтому для поиска повторяющихся слов, например, "the the" или "had had", можно применить такой шаблон:

```
/\b(\w+) \1\b/i
```

Но чаще всего вы будете прибегать к формату $1, потому что обычно шаблон применяется и с подстроками выполняются какие-то действия. Допустим, есть некоторый текст (почтовый заголовок), который выглядит так:

```
From: gnat@perl.com
To: camelot@oreilly.com
Date: Mon, 17 Jul 2000 09:00:00 -1000
Subject: Eye of the needle
```

и требуется создать хеш, в котором будет сопоставлен текст перед каждым двоеточием и текст после него. Если организовать построчный проход этого текста в цикле (например, при чтении его из файла), создать хеш можно так:

---

[1] Нельзя использовать $1 для ссылки на найденный текст внутри шаблона, поскольку эта переменная была бы интерполирована при компиляции регулярного выражения. Поэтому внутри шаблонов для ссылки на найденный текст мы применяем традиционное обозначение \1. Для двух- и трехзначных номеров ссылок возникает некоторая неоднозначность из-за сходства с восьмеричной нотацией символов, но она аккуратно разрешается при рассмотрении числа захваченных шаблонов. Например, для Perl метазнак \11 эквивалентен $11, только если ранее в шаблоне захвачено не менее 11 подстрок. В противном случае это эквивалентно \011, т. е. символу табуляции.

```
while (<>) {
    /^(.*?): (.*)$/;    # Текст до двоеточия - в $1, после него - в $2
    $fields{$1} = $2;
}
```

Подобно $`, $& и $' эти нумерованные переменные имеют динамическую область видимости до конца охватывающего блока или строки eval либо до следующего успешного поиска по шаблону, в зависимости от того, что произойдет раньше. Их можно также использовать в правой (заменяющей) части подстановки:

```
s/^(\S+) (\S+)/$2 $1/;  # Поменять местами первые два слова
```

Группирование может быть вложенным и в этом случае нумерация групп производится согласно положению левой скобки. Поэтому для строки «Primula Brandybuck» шаблон

```
/^((\w+) (\w+))$/
```

захватит "Primula Brandybuck" в $1, "Primula" в $2 и "Brandybuck" в $3. Это отражено на рис. 5.1.

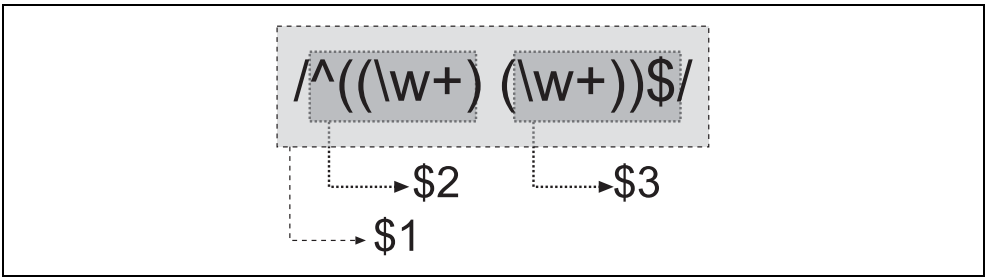


Рис. 5.1. Создание ссылок на найденный текст с помощью круглых скобок

Шаблоны с захватами часто применяются в списковом контексте для заполнения списка значениями, т. к. шаблон достаточно «сообразителен», чтобы возвращать перехваченные строки в виде списка:

```
($first, $last)        =  /^(\w+) (\w+)$/;
($full, $first, $last) =  /^((\w+) (\w+))$/;
```

При наличии модификатора /g шаблон может возвращать несколько подстрок от нескольких найденных соответствий в одном списке. Предположим, что почтовый заголовок, который мы видели выше, целиком находится в одной строке (скажем, в $_). То же самое, что в нашем построчном цикле, можно сделать с помощью одной команды:

```
%fields = /^(.*?): (.*)$/gm;
```

Шаблон находит соответствие четыре раза, и в каждом соответствии он находит две подстроки. Поиск с модификатором /gm возвращает их все в плос-

ком списке из восьми строк, которые присвоение списку %fields удачно проинтерпретирует как четыре пары ключ/значение, восстановив таким образом гармонию во вселенной.

Есть еще несколько специальных переменных, относящихся к тексту, захватываемому при поиске по шаблону. $& содержит всю найденную строку, $` содержит все, что находится слева от соответствия, $' – справа, а $+ содержит текст последней ссылки на найденный текст.

```
$_ = "Speak, <EM>friend</EM>, and enter.";
m[ (<.*?>) (.*?) (</.*?>) ]x;      # Тег, затем символы, затем закрывающий тег
print "prematch: $`\n";            # Speak,
print "match: $&\n";               # <EM>friend</EM>
print "postmatch: $'\n";           # , and enter.
print "lastmatch: $+\n";           # </EM>
```

Разъяснение этих магических переменных (и способ их написания по-английски) имеются в главе 28 «Специальные имена».

Массив @- (@LAST_MATCH_START) содержит смещения начал всех подчиненных соответствий, а массив @+ (@LAST_MATCH_END) содержит смещения концов:

```
#!/usr/bin/perl
$alphabet = "abcdefghijklmnopqrstuvwxyz";
$alphabet =~ /(hi).*(stu)/;

print "Все    соответствия начались в $-[0] и закончились в $+[0]\n";
print "Первое соответствие началось в $-[1] и закончилось в $+[1]\n";
print "Второе соответствие началось в $-[2] и закончилось в $+[2]\n";
```

Если в действительности требуется найти совпадение с литеральным символом скобок, а не интерпретировать его как метасимвол, поставьте перед ним обратную косую черту:

```
/\(e.g., .*?\)/
```

Это соответствует примеру в скобках (т. е. данному утверждению). Но поскольку точка является групповым символом, это соответствует и любому утверждению в скобках с первой буквой e и третьей буквой g (т. е. и этому утверждению).

Кластеризация

Голые скобки и группируют, и захватывают. Но иногда нам требуется не это. Иногда надо просто сгруппировать части шаблона, не создавая ссылки на найденный текст. Можно использовать расширенную форму скобок для подавления захвата: обозначение (?:*PATTERN*) ***кластеризует*** без захвата.

Есть по крайней мере три причины, по которым может потребоваться кластеризация без захвата:

1. Для квантифицирования чего-либо.
2. Чтобы ограничить область видимости внутреннего чередования; например, шаблон `/^cat|cow|dog$/` должен стать `/^(?:cat|cow|dog)$/` , чтобы кошка (cat) не убежала с `^`.
3. Чтобы ограничить область видимости модификатора шаблона конкретным подшаблоном, как, например, в `/foo(?-i:Case_Matters)bar/i`. (См. раздел «Замкнутые модификаторы шаблонов».)

Кроме того, подавление захвата того, что мы не собираемся использовать, повышает производительность. Отрицательной стороной является несколько более замусоренный внешний вид.

Применение в шаблоне левой круглой скобки и непосредственно за ней вопросительного знака означает *расширение* регулярного выражения. Бестиарий регулярных выражений в настоящее время относительно фиксирован, и мы не осмеливаемся создать новый метасимвол из опасения нарушить работу старых программ Perl. Вместо этого для добавления в бестиарий новых функций используется синтаксис расширения.

В оставшейся части этой главы мы увидим много других расширений регулярных выражений, которые кластеризуют без захвата, а также делают что-то еще. Особенность расширения `(?:PATTERN)` как раз в том, что оно ничего больше не делает. Поэтому высказывание:

```
@fields = split(/\b(?:a|b|c)\b/)
```

похоже на:

```
@fields = split(/\b(a|b|c)\b/)
```

но не возвращает ненужных полей. (Оператор `split` немного похож на `m//g` в том, что выдает лишние поля для всех строк, захваченных в шаблоне. Обычно `split` возвращает только то, с чем совпадений *не* найдено. Дополнительные сведения о `split` см. в главе 29.)

Замкнутые модификаторы шаблонов

Можно *замкнуть* (*cloister*) модификаторы `/i`, `/m`, `/s` и `/x`, ограничив их действие только частью шаблона, поместив их (без косой черты) между символами `?` и `:`, образующими обозначение кластера. Если сказать:

```
/Harry (?i:s) Truman/
```

то это будет соответствовать обеим строкам, `"Harry S Truman"` и `"Harry s Truman"`, в то время как:

```
/Harry (?x: [A-Z] \.? \s )?Truman/
```

соответствует "Harry S Truman" и "Harry S. Truman", а также "Harry Truman", а шаблон:

```
/Harry (?ix: [A-Z] \.? \s )?Truman/
```

соответствует пяти строкам, соединяя в замыкании модификаторы /i и /x.

Можно вычитать модификаторы из замыканий с помощью знака «минус»:

```
/Harry (?x-i: [A-Z] \.? \s )?Truman/i
```

Это соответствует любому выделению заглавными буквами в имени, но если есть средний инициал, он должен быть заглавной буквой, поскольку действие /i, примененного ко всему шаблону, отменяется внутри замыкания.

Опустив двоеточие и *PATTERN*, можно экспортировать установки модификаторов во внешний кластер, превращая его в замыкание. Это означает, что можно избирательно включать и выключать модификаторы для кластера, непосредственно охватывающего скобки модификаторов, например:

```
/(?i)foo/            # Эквивалентно /foo/i
/foo((?-i)bar)/i     # "bar" должно быть в нижнем регистре
/foo((?x-i) bar)/    # Включает /x и отключает /i для "bar"
```

Обратите внимание, что во втором и третьем примерах создаются ссылки на возвращаемый текст. Если они не нужны, следует применить (?-i:bar) и (?x-i: bar), соответственно.

Установка модификаторов для части шаблона особенно полезна, когда необходимо, чтобы . соответствовала символам перевода строки в некоторой части шаблона, но не во всем шаблоне. Установка /s для всего шаблона для этих целей бесполезна.

Чередование

Внутри шаблона или подшаблона метасимвол | служит для задания набора возможностей, любая из которых считается соответствием. Например:

```
/Gandalf|Saruman|Radagast/
```

соответствует Gandalf или Saruman, или Radagast. Чередование простирается до ближайшей закрывающей скобки (осуществляющей захват или нет):

```
/prob|n|r|l|ate/    # Соответствует prob, n, r, l или ate
/pro(b|n|r|l)ate/   # Соответствует probate, pronate, prorate или prolate
/pro(?:b|n|r|l)ate/ # Соответствует probate, pronate, prorate или prolate
```

Вторая и третья форма ищут одинаковые строки, но во второй форме переменный символ захватывается в $1, а в третьей – нет.

В каждой данной позиции машина регулярных выражений пытается найти соответствие с первой альтернативой, затем со второй и т. д. Длина альтернатив не имеет значения, поэтому в шаблоне:

```
/(Sam|Samwise)/
```

переменная $1 никогда не получит значения Samwise, в какой бы строке ни производился поиск, поскольку всегда сначала будет найдено соответствие Sam. Осуществляя такой перекрывающийся поиск, ставьте более длинные альтернативы в начало шаблона.

Но порядок альтернатив имеет значение только в данной позиции. Внешний цикл машины регулярных выражений осуществляет поиск слева направо, поэтому следующее выражение всегда соответствует первому Sam:

```
"'Sam I am,' said Samwise" =~ /(Samwise|Sam)/;   # $1 eq "Sam"
```

Однако можно принудительно вызвать просмотр справа налево, используя жадные квантификаторы, как обсуждалось ранее в разделе «Квантификаторы»:

```
"'Sam I am,' said Samwise" =~ /.*(Samwise|Sam)/; # $1 eq "Samwise"
```

Можно изменить порядок поиска слева направо (или справа налево) с помощью различных утверждений позиции, рассмотренных нами выше, таких как \G, ^ и $. Здесь мы привязываем шаблон к концу строки:

```
"'Sam I am,' said Samwise" =~ /(Samwise|Sam)$/;  # $1 eq "Samwise"
```

В этом примере $ выносится за скобки чередования (поскольку у нас уже есть парочка скобок, после которых можно его поместить), но и в отсутствие скобок можно расставить утверждения перед некоторыми или всеми имеющимися альтернативами соответственно тому, какое соответствие для них нам требуется. В следующей маленькой программе выводятся все строки, начинающиеся с маркеров __DATA__ или __END__:

```
#!/usr/bin/perl
while (<>) {
    print if /^__DATA__|^__END__/;
}
```

Но будьте при этом осторожны. Помните, что первая и последняя альтернативы (перед первой | и вслед за последней) стремятся схватить все элементы регулярного выражения с той или с другой стороны вплоть до конца выражения, если только нет охватывающих скобок. Часто встречается такая ошибка:

```
/^cat|dog|cow$/
```

когда в действительности имеется в виду:

```
/^(cat|dog|cow)$/
```

Первый шаблон хватает "cat" в начале строки или "dog" в любом месте, или "cow" в конце строки. Второй соответствует любой строке, состоящей только из "cat" или "dog", или "cow". Он также захватывает результат в $1, что может быть нам не нужно, тогда можно сказать:

```
/^cat$|^dog$|^cow$/
```

Ниже мы покажем другое решение.

Альтернатива может быть пустой и в этом случае она всегда соответствует.

```
/com(pound|)/;       # Соответствует "compound" или "com"
/com(pound(s|)|)/;   # Соответствует "compounds", "compound" или "com"
```

Это похоже на использование квантификатора ?, который соответствует 0 или 1 раз:

```
/com(pound)?/;       # Соответствует "compound" или "com"
/com(pound(s?))?/;   # Соответствует "compounds", "compound" или "com"
/com(pounds?)?/;     # То же, но не использует $2
```

Однако есть одно отличие. Когда ? применяется к шаблону, осуществляющему захват в нумерованную переменную, эта переменная остается неопределенной, если в нее не попадает никакая строка. При использовании нумерованной переменной она тоже ложная, но определена, и в ней находится нулевая строка.

Управление процессом

Всякому хорошему менеджеру известно, что не следует чрезмерно опекать своих подчиненных. Нужно просто указать им, что требуется, и позволить самим определять, как лучше всего этого добиться. Аналогично, часто лучше всего представлять себе регулярное выражение как своего рода указание: «Вот что мне нужно; пойди и найди мне подходящую строку».

С другой стороны, лучшие менеджеры также разбираются в той работе, которую пытаются сделать их подчиненные. То же самое верно в отношении поиска по шаблону в Perl. Чем глубже мы понимаем, как Perl решает задачу поиска по некоторому шаблону, тем разумнее мы сможем использовать возможности Perl для поиска по шаблону.

Один из наиболее важных моментов в поиске по шаблону в Perl заключается в том, чтобы понять, когда его применять *не* нужно.

Пусть Perl сделает дело

У людей с определенным темпераментом знакомство с регулярными выражениями вызывает соблазн любую задачу рассматривать как задачу поиска по шаблону. И хотя это может быть и верно в широком смысле, но поиск по шаблону – это нечто большее, чем простое вычисление регулярных выраже-

ний. Отчасти это означает искать ключи от машины там, где вы их уронили, а не под уличным фонарем, где лучше видно. И в повседневной жизни мы все понимаем, что первый вариант значительно более продуктивен.

Аналогично, нужно управлять последовательностью команд, выполняемых Perl для принятия решения о том, какие шаблоны выполнять, а какие – пропустить. Регулярные выражения довольно сообразительны, но это сообразительность лошади. Если она видит сразу слишком много, ее внимание может рассеяться. Поэтому иногда нужно надевать на нее шоры. Вспомните приведенный ранее пример чередования:

```
/Gandalf|Saruman|Radagast/
```

Он работает так, как обещано, но не так хорошо, как могло бы быть, поскольку в каждой позиции строки ищет все имена, прежде чем переместиться к следующей позиции. Внимательные читатели «Властелина колец» вспомнят, что из трех названных магов Гэндальф упоминается значительно чаще, чем Саруман, а Саруман упоминается значительно чаще, чем Радагаст. Поэтому обычно для осуществления чередования логические операторы Perl более эффективны:

```
/Гэндальф/ || /Саруман/ || /Радагаст/
```

Это еще один способ преодолеть «левый уклон» машины регулярных выражений. При этом Сарумана мы ищем только тогда, когда не видно Гэндальфа. А Радагаста – только тогда, когда Сарумана тоже нет.

Это позволяет не только изменить порядок поиска объектов, но и (иногда) улучшить работу оптимизатора регулярных выражений. Обычно проще оптимизировать поиск отдельной строки, чем нескольких строк одновременно. Аналогично, поиск с привязкой тоже часто можно оптимизировать, если он не слишком сложен.

Не обязательно ограничивать управление последовательностью выполняемых команд только оператором `||`. Часто можно осуществлять управление на уровне команд. Всегда следует подумать о том, чтобы сначала исключить общие случаи. Допустим, что вы пишете цикл для обработки файла конфигурации. Многие файлы конфигурации состоят, в основном, из комментариев. Часто лучше всего сразу выбросить комментарии и пустые строки, прежде чем приступать к интенсивной обработке, даже если эта интенсивная обработка выбрасывает комментарии и пустые строки по ходу дела:

```
while (<CONF>) {
    next if /^#/;
    next if /^\s*(#|$)/;
    chomp;
    munchabunch($_);
}
```

Даже если мы не стремимся к повышению производительности, часто требуется чередовать обычные выражения Perl с регулярными выражениями

просто потому, что необходимо выполнить какие-то действия, которые невозможно (или трудно) выполнить из регулярного выражения, например осуществить вывод. Вот полезный классификатор чисел:

```
warn "содержит нецифры"    if     /\D/;
warn "не натуральное число" unless /^\d+$/;             # отвергает -3
warn "не целое число"       unless /^-?\d+$/;           # отвергает +3
warn "не целое число"       unless /^[+-]?\d+$/;
warn "не десятичное число"  unless /^-?\d+\.?\d*$/;     # rejects .2
warn "не десятичное число"  unless /^-?(?:\d+(?:\.\d*)?|\.\d+)$/;
warn "не число с плавающей точкой С"
       unless /^([+-]?)(?=\d|\.\d)\d*(\.\d*)?([Ee]([+-]?\d+))?$/;
```

Можно было бы надолго растянуть этот раздел, но, в действительности, такого рода вещам посвящена вся книга. По ходу изложения мы увидим многочисленные примеры взаимодействия кода Perl и поиска по шаблону. В частности, имеется в виду раздел «Программные шаблоны» далее в этой главе. (Конечно, можно сначала прочесть материал, прерывающий изложение.)

Интерполяция переменных

Применение механизмов Perl, управляющих последовательностью выполняемых команд, для управления поиском с использованием регулярных выражений имеет свои ограничения. Основная сложность в том, что это подход типа «все или ничего»; либо мы выполняем шаблон, либо нет. Иногда общие черты требуемого шаблона ясны, но желательно иметь возможность его параметризации. Такую возможность предоставляет интерполяция переменных, подобно тому, как параметризация подпрограммы позволяет в большей мере оказывать влияние на ее выполнение, чем просто решать, вызывать ее или нет. (Подробнее о подпрограммах рассказано в следующей главе.)

Одно из элегантных применений интерполяции состоит в том, чтобы создать некоторую степень абстракции и при этом несколько повысить удобочитаемость. Посредством регулярных выражений можно, несомненно, добиться краткости записи:

```
if ($num =~ /^[-+]?\d+\.?\d*$/) { ... }
```

Но наши намерения становятся более очевидными, если записать так:

```
$sign = '[-+]?';
$digits = '\d+';
$decimal = '\.?';
$more_digits = '\d*';
$number = "$sign$digits$decimal$more_digits";
...
if ($num =~ /^$number$/o) { ... }
```

Мы еще поговорим о таком использовании интерполяции в разделе «Генерируемые шаблоны» далее в этой главе. Просто отметим, что при помощи

модификатора /o мы подавили повторную компиляцию шаблона, поскольку предполагаем, что $number не будет изменять свое значение во время выполнения программы.

Другой изящный прием состоит в том, чтобы вывернуть проверки «наизнанку» и задать переменную строку для поиска по шаблону в наборе известных заранее строк:

```
chomp($answer = <STDIN>);
if    ("SEND"  =~ /^\Q$answer/i) { print "Action is send\n"  }
elsif ("STOP"  =~ /^\Q$answer/i) { print "Action is stop\n"  }
elsif ("ABORT" =~ /^\Q$answer/i) { print "Action is abort\n" }
elsif ("LIST"  =~ /^\Q$answer/i) { print "Action is list\n"  }
elsif ("EDIT"  =~ /^\Q$answer/i) { print "Action is edit\n"  }
```

Этот код позволяет нам выполнять действие «send», вводя любую строку из S, SE, SEN или SEND (произвольно чередуя верхний и нижний регистры). Для действия «stop» нужно ввести, по крайней мере, ST (или St, или sT, или st).

Когда встречается обратная косая черта

Рассматривая интерполяцию строк в двойных кавычках, мы обычно имеем в виду интерполяцию переменных и обратной косой черты. Но, как уже говорилось, регулярные выражения обрабатываются за два прохода, и на проходе интерполяции интерпретация обратной косой черты, по большей части, откладывается для обработки анализатором регулярных выражений (который мы обсудим ниже). Обычно мы не замечаем разницы, поскольку Perl берет на себя труд ее скрыть. (Одной из последовательностей, которая очевидным образом ведет себя по-разному, является метазнак \b, который становится утверждением границы слова – во всяком случае, вне классов символов. Внутри класса символов, где утверждения не имеют смысла, он получает свое обычное значение символа возврата.)

В действительности очень важно, чтобы анализатор регулярных выражений обрабатывал символ обратной косой черты. Допустим, что мы выполняем поиск символов табуляции по шаблону с модификатором /x:

```
($col1, $col2) = /(.*?) \t+ (.*?)/x;
```

Если бы Perl не откладывал интерпретацию \t до анализатора регулярных выражений, то \t обратился бы в пробельный символ, который анализатор регулярных выражений и проигнорировал бы из-за модификатора /x. Но Perl не настолько подл или коварен.

Однако можно перехитрить самого себя. Предположим, что мы выделили разделитель колонок таким способом:

```
$colsep = "\t+";                          # (двойные кавычки)
($col1, $col2) = /(.*?) $colsep (.*?)/x;
```

Ну вот мы все и испортили, потому что \t превратится в действительный символ табуляции, прежде чем попадет на анализатор регулярных выраже-

ний, который будет думать, что мы сказали /(.*?)+(.*?)/, после того как выбросит пробельные символы. Чтобы исправить положение, не указывайте /x или используйте одинарные кавычки. Еще лучше, обратитесь к qr//. (См. следующий раздел.)

Единственными escape-последовательностями в двойных кавычках, обрабатываемыми в этом качестве, являются шесть последовательностей трансляции: \U, \u, \L, \l, \Q и \E. Если бы мы посмотрели, как устроен компилятор регулярных выражений Perl, то нашли бы в нем код для обработки таких escape-последовательностей, как \t для символа табуляции, \n для символа перевода строки и т. д. Но кода для обработки этих шести escape-последовательностей трансляции мы бы не нашли. (Мы перечислили их в табл. 5.7 только потому, что многие рассчитывают найти их там.) Если каким-то образом нам удастся протащить их в шаблон, не проведя сначала через вычисление в двойных кавычках, они не будут распознаны.

Как могут они прокрасться? Можно подавить интерполяцию путем использования одиночных кавычек в качестве ограничителя шаблона. В m'...', qr'...' и s'...'...' одинарные кавычки подавляют интерполяцию переменных и обработку escape-последовательностей, как если бы они находились в строках, заключенных в одинарные кавычки. Если сказать m'\ufrodo', то бедняга frodo (заглавными буквами) найден не будет. Однако, поскольку «обычные» символы обратной косой черты во всяком случае не обрабатываются на этом уровне, m'\t\d' все же соответствует действительному символу табуляции, за которым следует цифра.

Другим способом преодоления интерполяции служит сама интерполяция. Если сказать:

```
$var = '\U';
/${var}frodo/;
```

то несчастный frodo остается не выделенным заглавными буквами. Perl не станет повторять проход интерполяции лишь потому, что вы интерполировали нечто, что, похоже, может быть повторно интерполировано. Нельзя рассчитывать, что это будет работать лучше, чем такая двойная интерполяция:

```
$hobbit = 'Frodo';
$var = '$hobbit';           # (одинарные кавычки)
/$var/;                     # означает m'$hobbit', а не m'Frodo'.
```

Вот еще пример, показывающий, что большинство обратных косых черт интерпретируется анализатором регулярных выражений, а не при интерполяции переменных. Представим себе, что есть простая программа в стиле *grep*, написанная на Perl:[1]

---

[1] Если раньше вы не знали, что за программа *grep*, то теперь знаете. Программа *grep* должна быть в любой системе: мы считаем, что *grep* является самой полезной маленькой программой из когда-либо созданных. (Отсюда логически следует, что мы не считаем Perl маленькой программой.)

```
#!/usr/bin/perl
$pattern = shift;
while (<>) {
    print if /$pattern/o;
}
```

Если назвать эту программу *pgrep* и вызвать ее так:

```
% pgrep '\t\d' *.c
```

то мы обнаружим, что она выводит все строчки исходного кода на C, в которых за символом табуляции следует цифра. Нам не пришлось делать что-либо особенное, чтобы заставить Perl понять, что `\t` означает табуляцию. Если бы шаблоны Perl были просто интерполируемыми строками в двойных кавычках, нам бы пришлось; к счастью, они такими не являются. Они распознаются непосредственно анализатором регулярных выражений.

В действительности программа *grep* имеет ключ `-i`, который отключает поиск с учетом регистра. Этот ключ не нужно добавлять к вашей программе *pgrep*; она и так может обрабатывать его без модификации. Нужно лишь передать ей несколько более замысловатый шаблон со встроенным модификатором `/i`:

```
% pgrep '(?i)ring' LotR*.pod
```

Эта команда находит «Ring», «ring», «RING» и т. д. Эта функция не очень заметна в литеральных шаблонах, поскольку всегда можно просто написать `/ring/i`. Но для шаблонов, передаваемых в командной строке, в формах поиска в Интернете или встроенных в файлы конфигурации это может быть спасением. (Если говорить о кольцах.)

Оператор кавычек регулярных выражений qr//

Переменные, интерполирующиеся в шаблоны, делают это на этапе выполнения, а не компиляции. Это замедляет выполнение, поскольку Perl должен проверять, не изменил ли пользователь содержимое переменной; если да, то перекомпилирует регулярное выражение. Как указывалось в разделе «Операторы поиска по шаблону», если мы обещаем не изменять шаблон, то можем использовать опцию `/o`, чтобы интерполировать и компилировать только один раз:

```
print if /$pattern/o;
```

Хотя это и работает в нашей программе *pgrep*, но в общем случае это не работает. Предположим, у нас есть куча шаблонов и мы собираемся искать их все в цикле, возможно, так:

```
foreach $item (@data) {
    foreach $pat (@patterns) {
        if ($item =~ /$pat/) { ... }
    }
}
```

Нельзя написать /$pat/o, поскольку значение $pat изменяется при каждом прохождении внутреннего цикла.

Решением является оператор qr/*PATTERN*/imosx, который заключает в кавычки и компилирует как регулярное выражение передаваемый ему шаблон *PATTERN*. *PATTERN* интерполируется таким же образом, как в m/*PATTERN*/. Если в качестве ограничителя используется одинарная кавычка ', то интерполяция переменных (и шести escape-последовательностей трансляции) не производится. Оператор возвращает значение Perl, которое можно использовать вместо эквивалентного литерала в соответствующем поиске по шаблону или подстановке. Например,

```
$regex = qr/my.STRING/is;
s/$regex/something else/;
```

эквивалентно:

```
s/my.STRING/something else/is;
```

Поэтому в приведенной выше задаче с вложением цикла обработайте предварительно свой шаблон в отдельном цикле:

```
@regexes = ();
foreach $pat (@patterns) {
    push @regexes, qr/$pat/;
}
```

Или разом с помощью оператора Perl map:

```
@regexes = map { qr/$_/ } @patterns;
```

А затем измените цикл, чтобы он использовал прекомпилированные регулярные выражения:

```
foreach $item (@data) {
    foreach $re (@regexes) {
        if ($item =~ /$re/) { ... }
    }
}
```

Теперь при запуске поиска Perl не должен будет создавать компилированное регулярное выражение при каждой проверке if, потому что он видит, что оно у него уже есть.

Результат qr// можно даже интерполировать в более крупный поиск, как если бы он был простой строкой:

```
$regex = qr/$pattern/;
$string =~ /foo${regex}bar/;   # интерполировать в более крупный шаблон
```

На этот раз Perl перекомпилирует шаблон, но всегда можно связать вместе несколько операторов qr//.

Это работает, потому что оператор qr// возвращает объект особого типа, в котором перегружена операция превращения в строку, как это описано в главе 13 «Перегрузка». Если вывести возвращаемое значение, то можно увидеть эквивалентную строку:

```
$re = qr/my.STRING/is;
print $re;                  # выводит (?si-xm:my.STRING)
```

Модификаторы /s и /i включены в шаблоне, потому что они были переданы qr//. Модификаторы /x и /m, однако, отключены, потому что не были переданы.

Всякий раз, интерполируя в шаблон строки неизвестного происхождения, следует быть готовым к обработке исключительных ситуаций, возбуждаемых компилятором регулярных выражений, если кто-то подбросил вам строку с дикими зверюшками:

```
$re = qr/$pat/is;                       # может убежать и съесть вас
$re = eval { qr/$pat/is } || warn ...   # попалась в наружную клетку
```

Подробнее об операторе eval см. главу 29.

Компилятор регулярных выражений

После того как интерполирующий проход сделает со строкой свое дело, наконец-то возможность разобраться с ней предоставляется анализатору регулярных выражений. Фактически в этом месте может возникнуть не очень много неприятностей, если только нет путаницы в скобках и не используются ничего не значащие последовательности метасимволов. Анализатор осуществляет рекурсивно-нисходящий анализ регулярного выражения, и если разбор произведен, оно превращается в формат, пригодный для интерпретации машиной регулярных выражений (см. следующий раздел). Самое интересное из происходящего в анализаторе – это оптимизация регулярного выражения для максимально быстрого его выполнения. Эту часть мы излагать не будем. Это профессиональный секрет. (Слухи о том, что, глядя на код, обрабатывающий регулярные выражения, можно сойти с ума, сильно преувеличены. Как мы надеемся.)

Но можно поинтересоваться, что анализатор думает о нашем регулярном выражении, и если вежливо его об этом спросить, он ответит. Сказав use re "debug", можно увидеть, как анализатор регулярных выражений обрабатывает шаблон. (Те же данные можно получить, задав ключ командной строки -Dr, которым можно пользоваться, если при установке Perl вы компилировали его с флагом -DDEBUGGING.)

```
#!/usr/bin/perl
use re "debug";
"Smeagol" =~ /^Sm(.*)g[aeiou]l$/;
```

Ниже приведен вывод. Как можно видеть, перед выполнением Perl компилирует регулярное выражение и назначает смысл компонентам шаблона: BOL для начала строки (^), REG_ANY для точки и т. д.:

```
Compiling REx `^Sm(.*)g[aeiou]l$'
size 24 first at 2
rarest char l at 0
rarest char S at 0
   1: BOL(2)
   2: EXACT <Sm>(4)
   4: OPEN1(6)
   6:   STAR(8)
   7:     REG_ANY(0)
   8: CLOSE1(10)
  10: EXACT <g>(12)
  12: ANYOF[aeiou](21)
  21: EXACT <l>(23)
  23: EOL(24)
  24: END(0)
anchored `Sm' at 0 floating `l'$ at 4..2147483647
     (checking anchored) anchored(BOL) minlen 5
Omitting $` $& $' support.
```

Некоторые строки содержат выводы, сделанные оптимизатором регулярных выражений. Ему известно, что строка должна начинаться с "Sm", и потому нет оснований для обычного просмотра слева направо. Ему известно, что строка должна оканчиваться на "l", поэтому иные строки он без промедления отвергает. Ему известно, что строка должна быть не короче пяти символов, поэтому более короткую строку он игнорирует с лету. Ему также известно, какой символ самый редкий в каждой константной строке, что иногда помогает при поиске в «изученных» строках. (См. study в главе 29.)

Затем производится трассировка выполнения поиска по шаблону:

```
EXECUTING...

Guessing start of match, REx `^Sm(.*)g[aeiou]l$' against `Smeagol'...
Guessed: match at offset 0
Matching REx `^Sm(.*)g[aeiou]l$' against `Smeagol'
  Setting an EVAL scope, savestack=3
   0 <> <Smeagol>         |  1:  BOL
   0 <> <Smeagol>         |  2:  EXACT <Sm>
   2 <Sm> <eagol>         |  4:  OPEN1
   2 <Sm> <eagol>         |  6:  STAR
                           REG_ANY can match 5 times out of 32767...
  Setting an EVAL scope, savestack=3
   7 <Smeagol> <>         |  8:    CLOSE1
   7 <Smeagol> <>         | 10:    EXACT <g>
                              failed...
   6 <Smeago> <l>         |  8:    CLOSE1
   6 <Smeago> <l>         | 10:    EXACT <g>
```

```
                                  failed...
    5 <Smeag> <ol>          |  8:    CLOSE1
    5 <Smeag> <ol>          | 10:    EXACT <g>
                                  failed...
    4 <Smea> <gol>          |  8:    CLOSE1
    4 <Smea> <gol>          | 10:    EXACT <g>
    5 <Smeag> <ol>          | 12:    ANYOF[aeiou]
    6 <Smeago> <l>          | 21:    EXACT <l>
    7 <Smeagol> <>          | 23:    EOL
    7 <Smeagol> <>          | 24:    END
Match successful!
Freeing REx: `^Sm(.*)g[aeiou]l$`
```

Если взглянуть на пробелы в середине `Smeagol`, то можно увидеть, как машина регулярных выражений делает перелет, чтобы дать возможность `.*` проявить максимальную жадность, а затем откатывается назад, пока не станет возможным уместить оставшуюся часть шаблона. Но об этом в следующем разделе.

Маленькая машинка, которая /(не) может?/

А теперь мы хотим рассказать сказку о маленьком Регексе – машине регулярных выражений, которая говорит: «Думаю, что смогу. Думаю, что смогу. Думаю, что смогу».

В этом разделе мы излагаем правила, которыми руководствуется машина регулярных выражений в Perl для поиска шаблона в строке. Машина регулярных выражений очень настойчива и трудолюбива. Вполне может оказаться, что она все еще работает, когда мы думаем, что она уже закончила работу. Машина регулярных выражений не сдастся, пока не будет полностью уверена, что найти соответствие между шаблоном и строкой невозможно. Приведенные ниже Правила объясняют, как машина регулярных выражений «думает, что сможет» как можно дольше, пока не будет *знать*, может она или нет. Проблема для нашей машины регулярных выражений в том, что ее задача не решается просто грубой силой. Она должна произвести поиск в потенциально очень сложном пространстве возможностей, следя при этом за тем, где она уже была, а где еще нет.

Машина регулярных выражений использует для поиска совпадения недетерминированный конечный автомат (НДКА). Это означает, что она следит за тем, что она опробовала, а что – нет, и если что-то не клеится, она возвращается и пробует что-то еще. Это известно как *перебор с возвратом* (*backtracking*). Машина регулярных выражений может перепробовать миллион подшаблонов в каком-либо месте, затем бросить их все, вернуться назад к некоторому варианту в начале и снова перебрать миллион подшаблонов в другой точке. Машина регулярных выражений не слишком умна, просто настойчива и дотошна. Если вы достаточно хитры, то дадите машине регулярных выражений эффективный шаблон, который не заставит ее делать много дурацких отступлений.

Когда кто-то бросает фразу типа «регулярные выражения выбирают самое левое, самое длинное совпадение», это означает, что Perl обычно предпочитает самое левое совпадение самому длинному. Но машина регулярных выражений не сознает, что она что-то «предпочитает», она вообще не думает, а просто потрошит шаблон. Общие предпочтения возникают как результат многих отдельных и не связанных между собой выборов вариантов. Вот они:[1]

Правило 1

Машина пытается найти совпадение в строке возможно левее, так чтобы регулярное выражение в целом находило соответствие согласно Правилу 2.

Машина начинает работу прямо перед первым символом и пытается найти соответствие всему шаблону с этой точки. Весь шаблон соответствует тогда и только тогда, когда машина достигает конца шаблона раньше, чем она соскочит с конца строки. Если совпадение найдено, она сразу прекращает работу; продолжать поиск «более удачного» совпадения она не станет, даже если шаблон может быть приложен многими различными способами.

Если она не может найти совпадение с шаблоном, начиная с первой позиции строки, то признает временное поражение и перемещается к следующей позиции в строке, между первым и вторым символами, и снова опробует все возможности. В случае успеха она останавливается. В случае неудачи продолжается движение по строке. Поиск в целом не считается безуспешным, пока не опробовано все регулярное выражение в каждой позиции строки, в том числе за последним символом.

В строке из n символов фактически имеется $n + 1$ позиция для поиска соответствия. Дело в том, что начало и конец соответствия находятся *между* символами строки. Это правило иногда удивляет тех, кто пишет шаблон типа `/x*/`, который может соответствовать нулю или более символов `"x"`. Если применить этот шаблон к строке `"fox"`, он не найдет `"x"`. Вместо этого он немедленно найдет совпадение с нулевой строкой перед `"f"` и больше ничего искать не станет. Для того чтобы найти один или более символов `x`, нужно использовать шаблон `/x+/`. О квантификаторах читайте в Правиле 5.

Следствием этого правила является то, что каждый шаблон, соответствующий нулевой строке, будет гарантированно соответствовать самой левой позиции строки (при отсутствии противоречащих этому утверждений нулевой ширины).

---

[1] Некоторые из этих вариантов могут быть пропущены, если оптимизатор регулярных выражений сказал свое слово, как если бы наша машинка проскочила гору с помощью квантового туннельного эффекта. Но в данном изложении об оптимизаторе нужно забыть.

Правило 2

Когда машина обнаруживает набор альтернатив (разделяемых символами |) – либо на верхнем уровне, либо на текущем уровне кластера, – она опробует их слева направо, останавливаясь на первом успешном совпадении, позволяющем успешно завершить поиск шаблона в целом.

Набор альтернатив соответствует строке, если соответствует какая-либо из альтернатив согласно Правилу 3. Если ни одна из альтернатив не соответствует, происходит возвращение к Правилу, вызвавшему данное Правило, которым обычно является Правило 1, но может также быть Правило 4 или 6, если мы находимся внутри кластера. Это правило ищет новую позицию, в которой нужно применить Правило 2.

Если альтернатива только одна, то либо она соответствует, либо нет, и Правило 2 по-прежнему применимо. (Такой вещи, как нулевая альтернатива, не существует, поскольку с нулевой строкой всегда устанавливается соответствие.)

Правило 3

Данная конкретная альтернатива соответствует строке, если все элементы, перечисленные в альтернативе, последовательно находятся в строке согласно Правилам 4 и 5 (так, чтобы при этом удовлетворялось регулярное выражение в целом).

Элемент состоит из *утверждения* (о них говорится в Правиле 4) или *квантифицированного атома* (о них говорится в Правиле 5). Элементам, для которых есть несколько вариантов соответствия, дается «порядок клева» слева направо. Если нельзя найти соответствие элементам в заданном порядке, то машина отступает и выбирает следующую альтернативу согласно Правилу 2.

Элементы, которые должны быть последовательно найдены, не разделяются в регулярном выражении с помощью какого-либо синтаксиса: они просто располагаются рядом в том порядке, в котором должны совпасть в строке. Когда мы просим найти `/^foo/`, то фактически просим найти четыре элемента, располагающиеся один за другим. Первый является утверждением нулевой ширины, отыскиваемым согласно Правилу 4, а остальные три представляют собой обычные символы, которые должны совпасть сами с собой, один за другим, согласно Правилу 5.

Порядок слева направо означает, что в шаблоне:

```
/x*y*/
```

сначала `x*` выбирает один способ соответствия, а затем `y*` перепробует все свои способы. Если последнее завершается неудачей, то `x*` получает возможность выбрать свой второй вариант и заставить `y*` снова перепробовать все свои способы, и т. д. Находящиеся правее элементы «изменяются быстрее», если позаимствовать выражение из многомерных массивов.

Правило 4

Если утверждению нет соответствия в текущей позиции, машина возвращается к Правилу 3 и осуществляет опробование других вариантов выбора для элементов более высокого порядка.

Некоторые утверждения более замысловаты, чем другие. Perl поддерживает много расширений регулярных выражений, из которых какие-то являются утверждениями нулевой ширины. Например, положительное заглядывание вперед `(?=...)` и отрицательное заглядывание вперед `(?!...)` не соответствуют в действительности никаким символам, а просто утверждают, что регулярное выражение, представленное `...`, должно (или не должно) соответствовать в этой точке, если бы мы гипотетически попробовали его применить.[1]

Правило 5

Атом с квантификатором соответствует строке, только если сам атом соответствует некоторое число раз, допускаемое квантификатором. (Соответствие самого атома устанавливается согласно Правилу 6.)

Для разных квантификаторов требуется разное число совпадений, и для большинства из них допускается диапазон числа совпадений. Многократные совпадения должны происходить подряд, т. е. примыкать в строке друг к другу. У неквантифицированного атома предполагается наличие квантификатора, требующего ровно одного совпадения (т. е. `/x/` означает то же самое, что `/x{1}/`). Если в текущей позиции нельзя найти совпадение с каким-либо допустимым количеством рассматриваемого атома, то машина возвращается к Правилу 3 и осуществляет опробование других вариантов выбора для элементов более высокого порядка.

Квантификаторами являются `*`, `+`, `?`, `*?`, `+?`, `??` и различные виды фигурных скобок. Если используется формат `{COUNT}`, то выбора нет, и атом должен совпадать ровно указанное число раз. В противном случае для числа совпадений атома имеется некоторый диапазон, и машина отслеживает все выбираемые варианты, чтобы при необходимости вернуться назад. Но тогда возникает вопрос о том, с которого из вариантов следует начать. Можно начать с максимального числа и затем снижать его, а можно – с минимального и увеличивать его.

Обычные квантификаторы (без вопросительного знака в конце) задают *жадный* поиск; это значит, что они пытаются найти совпадение, состоящее из возможно большего числа символов. Чтобы найти самое жадное соответствие, машине приходится проявлять некоторую осторожность. Неверное предположение может обойтись очень дорого, поэтому на прак-

[1] На самом деле машина действительно пытается найти соответствие. Машина возвращается к Правилу 2, чтобы проверить подшаблон, а затем стирает все упоминания о том, какая часть строки была «проглочена», и возвращает только значение утверждения в виде успеха или неудачи. (Захваченные подстроки, однако, запоминаются.)

тике машина не производит обратный отсчет от самого большого значения, которое может быть очень большим и повлечь миллионы неправильных предположений. На самом деле машина поступает несколько умнее: сначала она считает, сколько совпадающих атомов (подряд) в действительности имеется в строке, а затем использует этот фактический максимум в качестве первого выбранного значения. (Она также запоминает все более короткие варианты на случай, когда самый длинный не вписывается.) Затем она (в конце концов) пытается найти соответствие для оставшейся части шаблона, предполагая, что самый длинный вариант является самым лучшим. Если для самого длинного варианта оказывается невозможно найти соответствие с оставшейся частью шаблона, происходит откат и опробуется следующий по длине вариант.

Если, например, мы говорим `/.*foo/`, то машина пытается найти максимальное число «любых» символов (представляемое точкой) вплоть до конца строки, прежде чем пытаться найти `"foo"`; и когда для `"foo"` там не находится соответствия (и не может найтись, потому что в конце строки для него недостаточно места), машина начинает отдавать по одному символу, пока не найдет `"foo"`. Если `"foo"` встречается в строке более одного раза, машина остановится на последнем, поскольку в действительности это будет *первое* найденное при поиске с возвратом совпадение. Когда найден весь шаблон при некоторой определенной длине `.*`, машина знает, что может теперь отбросить все более короткие варианты для `.*` (те, которые она стала бы использовать, если бы текущее `"foo"` не подошло).

Помещая вопросительный знак вслед за жадным квантификатором, мы превращаем его в умеренный квантификатор, который для первой попытки выбирает самое маленькое количество. Поэтому, если сказать `/.*?foo/`, то `.*?` сначала пытается найти соответствие 0 символов, затем 1 символу, затем 2 и т. д., пока не сможет найти соответствие `"foo"`. Вместо того чтобы отступать назад, машина, так сказать, отступает вперед и заканчивает, найдя в строке первое `"foo"`, а не последнее.

Правило 6

Поиск каждого атома производится в соответствии с семантикой, предписанной для его типа. Если соответствия атому не найдено (или найденное не позволяет найти оставшуюся часть шаблона), то машина откатывается к Правилу 5 и опробует другой вариант количества атома.

Соответствие атомов определяется согласно следующим типам:

- Регулярное выражение в круглых скобках, `(...)`, соответствует тому же, чему соответствует регулярное выражение, представленное `(...)` согласно Правилу 2. Скобки действуют, как оператор кластеризации для квантифицирования. Скобки имеют также побочный эффект, захватывая найденную подстроку, которую можно потом использовать в *ссылке на найденный текст* (*backreference*). Побочный эффект можно подавить, используя конструкцию `(?:...)`, имеющую только кластеризующую семантику: она ничего не запоминает в `$1`, `$2` и т. д.

Возможны и другие формы атомов в скобках (и утверждений) – см. оставшийся материал этой главы.

- Точка соответствует любому символу, за исключением, возможно, символа перевода строки.
- Список символов в квадратных скобках (*класс символов*) соответствует любому из символов, указанных в списке.
- Буква с обратной косой чертой перед ней сответствует конкретному символу или символу из числа приведенных в табл. 5.7.
- Любой другой символ с обратной косой чертой перед ним соответствует этому символу.
- Любой символ, не упомянутый выше, соответствует самому себе.

Все это выглядит довольно сложным, но вывод из этого заключается в следующем. Для каждого набора вариантов, определяемого квантификатором или чередованием, у машины регулярных выражений есть ручка, которую можно покрутить. Она будет крутить эти ручки, пока не найдется соответствие шаблону в целом. Правила просто говорят, в каком порядке машине регулярных выражений позволено крутить эти ручки.

Слова «машина регулярных выражений предпочитает самое левое соответствие» просто означают, что ручку начальной позиции она крутит медленнее всего. А поиск с возвратом означает, что нужно открутить назад ручку, которую мы только что крутили, и попытаться покрутить ручку, находящуюся выше в порядке старшинства, положение которой изменяется медленней.

Вот более конкретный пример – программа, которая обнаруживает, когда конец одного из двух последовательных слов совпадает с началом другого:

```
$a = 'nobody';
$b = 'bodysnatcher';
if ("$a $b" =~ /^(\w+)(\w+) \2(\w+)$/) {
    print "$2 перекрывается в $1-$2-$3\n";
}
```

Программа выводит:

```
body перекрывается в no-body-snatcher
```

Можно предположить, что `$1` из-за своей жадности схватит все слово `"nobody"`. Так и происходит – поначалу. Но когда это сделано, не остается символов для переменной `$2`, которые необходимо поместить в нее из-за квантификатора `+`. Поэтому машина регулярных выражений отступает, и `$1` неохотно отдает один символ для `$2`.

На этот раз успешно отыскивается пробел, но затем машина видит `\2`, которая представляет жалкую букву `y`. Следующий символ в строке не `y`, а `b`. Машине регулярных выражений приходится постоянно отступать и делать еще

несколько попыток, в конце концов, заставив $1 отдать «body» переменной $2. Habeas corpus,[1] так сказать.

На самом деле, это не вполне срабатывает, когда само перекрытие является удвоением, как, например, в паре слов "rococo" и "cocoon". Приведенный выше алгоритм решит, что перекрывающейся строкой, $2, будет "co", а не "coco". Но нам не нужно "rococococoon"; нам нужно "rococoon". Это тот случай, когда вы можете оказаться умнее машины регулярных выражений. Добавив квантификатор минимального соответствия к части $1, получим значительно лучший шаблон /^(\w+?)(\w+) \2(\w+)$/, который делает как раз то, что нужно.

Значительно более подробное обсуждение достоинств и недостатков различных типов машин регулярных выражений можно найти в книге Джеффри Фридла (Jeffrey Friedl) «Mastering regular expressions». Машина регулярных выражений Perl работает очень хорошо при решении многих повседневных задач, которые требуется решать с помощью Perl, и даже не столь повседневных, если проявить к ней чуточку уважения и понимания.

Сложные шаблоны

Заглядывающие утверждения

Иногда требуется только глянуть украдкой. Есть четыре расширения регулярных выражений, которые позволяют сделать это и которые мы называем *заглядывающими утверждениями* (*lookaround assertions*), поскольку они позволяют выполнить своего рода разведку, не осуществляя настоящий поиск символов. Эти утверждения утверждают, что соответствие некоторому шаблону было бы (или не было) найдено, если бы мы попытались это сделать. Машина регулярных выражений организует это, осуществляя реальный поиск соответствия гипотетическому шаблону и делая потом вид, что оно не найдено (если найдено).

Когда машина регулярных выражений смотрит вперед из текущей позиции строки, мы называем это утверждением с *заглядыванием вперед* (*lookahead assertion*). Если она смотрит назад – утверждением с *заглядыванием назад* (*lookbehind assertion*). Шаблоны для заглядывания вперед могут быть любыми регулярными выражениями, но шаблоны для заглядывания назад могут иметь только фиксированную ширину, поскольку должны знать, откуда начинать свое гипотетическое соответствие.

Хотя все эти четыре расширения являются утверждениями нулевой ширины и потому не поглощают символов (по крайней мере, официально), на са-

[1] Акт «Habeas Corpus», гарантирующий свободу личности, принятый в Англии в 1679 г., согласно которому судебная власть обязана по требованию заинтересованных лиц обеспечить доставку в суд всякого, кто подвергся заключению, для проверки законности ареста. – *Примеч. ред.*

мом деле можно захватывать в них строки, если добавить дополнительные уровни захватывающих скобок.

(?=*PATTERN*) (утвердительное заглядывание вперед)

Когда машина встречает (?=*PATTERN*), она заглядывает дальше в строку и убеждается, что в ней встречается шаблон *PATTERN*. Если вы вспомните, то в нашей программе для удаления дубликатов слов пришлось написать цикл, поскольку шаблон каждый раз съедал слишком много:

```
$_ = "Paris in THE THE THE THE spring.";

удалить слова, повторяющиеся дважды (и трижды (и четырежды...))
1 while s/\b(\w+) \1\b/$1/gi;
```

Услышав слова «съел слишком много», всегда нужно подумать «утверждение с заглядыванием вперед». (Ну, почти всегда.) Заглядывая вперед, вместо того, чтобы проглотить второе слово, можно написать программу, удаляющую дубликаты за один проход, например:

```
s/ \b(\w+) \s (?= \1\b ) //gxi;
```

Конечно, это не совсем правильно, поскольку может исказить и допустимые фразы типа «The clothes you DON DON't fit».

(?!*PATTERN*) (отрицательное заглядывание вперед)

Когда машина встречает (?=*PATTERN*), она заглядывает дальше в строку и убеждается, что шаблон *PATTERN* в ней *не* встречается. Чтобы исправить наш предшествующий пример, можно добавить отрицательное утверждение с заглядыванием вперед после положительного утверждения и исключить случай сокращенной формы:

```
s/ \b(\w+) \s (?= \1\b (?! '\w))//xgi;
```

Это \w в конце необходимо, чтобы избежать смешивания со словами в конце строк в одинарных кавычках. Можно пойти еще дальше, чтобы законные обороты типа «that that particular» наша программа не «исправляла». Мы добавим альтернативу для отрицательного заглядывания вперед, чтобы предотвратить исправление «that» (при этом демонстрируя, что для кластеризации альтернатив может использоваться любая пара круглых скобок):

```
s/ \b(\w+) \s (?= \1\b (?! '\w | \s particular))//gix;
```

Теперь мы знаем, что фраза «that that particular» является безопасной. К несчастью, Геттисбергская речь[1] по-прежнему будет разрушена. Поэтому добавим еще одно исключение:

```
s/ \b(\w+) \s (?= \1\b (?! '\w | \s particular | \s nation))//igx;
```

[1] Речь Авраама Линкольна, произнесенная в 1863 году и считающаяся образцом ораторского искусства. Содержит оборот «that that nation» – «чтобы эта нация». – *Примеч. перев.*

Это уже начинает выходить из-под контроля. Поэтому составим Официальный список исключений и прибегнем к хитрому трюку с интерполяцией, чтобы переменная $" разделяла альтернативы символом |:

```
@thatthat = qw(particular nation);
local $" = '|';
s/ \b(\w+) \s (?= \1\b (?! '\w | \s (?: @thatthat )))//xig;
```

(?<=*PATTERN*) (утвердительное заглядывание назад)

Когда машина обнаруживает (?<=*PATTERN*), она смотрит назад в строку и проверяет, что шаблон *PATTERN* уже встретился.

В нашем примере все еще остается проблема. Хотя он уже позволяет Честному Эйбу произносить такие слова, как «that that nation», он также допускает фразы типа «Paris, in the the nation of France». Мы можем добавить положительное утверждение с заглядыванием назад перед нашим списком исключений, чтобы гарантировать применение исключений @thatthat действительно только к «that that».

```
s/ \b(\w+) \s (?= \1\b (?! '\w | (?<= that) \s (?: @thatthat )))//ixg;
```

Да, это становится ужасно сложным, но потому наша глава и названа «Сложные шаблоны», в конце концов. Если кому-то необходимо сделать шаблон еще более сложным, чем это нам пока удалось, то разумное использование комментариев и qr// поможет ему сохранить разум. Или, по крайней мере, в большей степени сохранить.

(?<!*PATTERN*) (отрицательное заглядывание назад)

Обнаружив (?<!=*PATTERN*), машина смотрит назад в строку и проверяет, что шаблон *PATTERN* не встретился.

На этот раз возьмем совсем простой пример. Простой вариант старого орфографического правила: «I перед E, если только не после C». На Perl ошибки исправляются так:

```
s/(?<!c)ei/ie/g
```

Программист должен сам решить для себя, будете ли он обрабатывать какие-либо исключения.

Подшаблоны без возврата

Как описывалось в «Маленькая машинка, которая /(не) могла?/», машина регулярных выражений часто осуществляет перебор с возвратом при прохождении шаблона. Можно помешать машине регулярных выражений осуществлять возврат к некоторому набору вариантов путем создания *подшаблона без возврата* (*nonbacktracking subpattern*). Подшаблон без возврата выглядит как (?>*PATTERN*) и работает так же, как простой (?:*PATTERN*), но если *PATTERN* находит соответствие, то возврат к другим квантификаторам или альтернативам внутри подшаблона подавляется. (Следовательно, бес-

смысленно использовать его с шаблоном, не содержащим квантификаторов или альтернатив.) Единственный способ заставить его изменить свое решение – это возвратиться к чему-либо до подшаблона и снова войти в подшаблон слева.

Это все равно, что покупать автомобили. После некоторой торговли вы ставите ультиматум: «Вот мое последнее предложение; либо оно принимается, либо мы расходимся». Если оно не принимается, вы не начинаете снова торговаться. Вместо этого вы осуществляете возврат через входную дверь. Можно пойти к другому дилеру и опять начать торговаться. Вам позволено снова торговаться, но только потому, что вы снова вошли в шаблон без возврата в другом контексте.

Приверженцы Prolog или SNOBOL могут представить это себе как оператор отсечения или ловушки с видимостью (scoped cut or fence operator).

Посмотрите, как в `"aaab" =~ /(?:a*)ab/` подшаблон `a*` сначала находит три `a`, а потом отдает одну из них, поскольку последняя `a` понадобится позже. Подгруппа жертвует частью того, что ей требуется, чтобы успешным оказался поиск в целом. (Что выглядит как позволение продавцу автомобиля уговорить вас отдать ему часть ваших денег, поскольку вы боитесь, что сделка не состоится.) Напротив, подшаблон в `"aaab" =~ /(?>a*)ab/` никогда не отдаст того, что схватил, даже если такое его поведение приведет к неудаче поиска в целом.

Хотя `(?>PATTERN)` полезен для изменения режима работы шаблона, он применяется в основном для того, чтобы ускорить получение неуспеха при некоторых видах поиска, когда известно, что он все равно окажется неудачным (если только не будет успешным сразу). Машине регулярных выражений может потребоваться весьма много времени, чтобы выйти на неуспех, особенно при вложенности квантификаторов. Следующий шаблон почти мгновенно приводит к успеху:

```
$_ = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab";
/a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*[b]/;
```

Но проблема не в успехе. Проблема в неуспехе. Если удалить из строки последнюю букву `b`, то шаблон, возможно, будет выполняться долгие и долгие годы, прежде чем будет получен неуспех. Многие и многие тысячелетия. Фактически миллиарды и миллиарды лет.[1] Внимательно посмотрев, мы обнаружим, что этот шаблон не может быть успешно найден, если в конце строки нет `b`, но оптимизатор регулярных выражений не настолько сообразителен (на момент написания книги), чтобы определить, что `/[b]/` эквива-

[1] На самом деле это скорее септиллионы и септиллионы. Мы не можем сказать точно, сколько времени потребуется. Мы не стали дожидаться результата. В любом случае, компьютер откажет раньше, чем случится тепловая смерть Вселенной, а это регулярное выражение закончит свою работу позже, чем любое из этих событий.

лентно /b/. Но если дать ему намек, то можно заставить его быстро определить неуспех, в то же время позволяя получить успех, когда это возможно:

```
/(?>a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*)[b]/;
```

Возьмем более реалистический (надеемся) пример. Представим себе программу, которая должна считывать сразу абзац и показывать строки, имеющие продолжение, которые отмечаются обратной косой чертой в конце строки. Вот образец из *Makefile* для Perl, в котором действует данное соглашение по продолжению строк:

```
# Files to be built with variable substitution before miniperl
# is available.
sh = Makefile.SH cflags.SH config_h.SH makeaperl.SH makedepend.SH \
        makedir.SH myconfig.SH writemain.SH
```

Можно написать такую простую программу:

```
#!/usr/bin/perl -00p
while ( /( (.+) ( (?<=\\) \n .* )+ ) /gx) {
    print "GOT $.: $1\n\n";
}
```

Она работает правильно, но очень медленно. Дело в том, что машина регулярных выражений отступает от конца строки с шагом в один символ, сокращая то, что находится в $1. Это неумно. Если избавиться от ненужного захвата, то это не сильно поможет. Использование в качестве шаблона:

```
(.+(?:(?<=\\)\n.*)+)
```

несколько ускоряет работу, но незначительно. В такой ситуации существенно помогает шаблон без возврата. Шаблон:

```
((?>.+)(?:(?<=\\)\n.*)+)
```

делает то же самое, но на порядок быстрее, поскольку не тратит время на возврат в поисках того, чего нет.

С помощью (?>...) нельзя добиться такого успеха, которого нельзя добиться с помощью (?:...) и даже простого (...). Но если провал неизбежен, лучше пусть он произойдет скорее, чтобы можно было жить дальше.

Программные шаблоны

Большинство программ на Perl следует императивному (называемому также процедурным) стилю программирования, в котором ряд разрозненных команд подается в легко обозримом порядке: «Разогреть плиту, перемешать, глазировать, нагреть, остудить, подать пришельцам.» Иногда в эту смесь добавляют чуть-чуть функционального программирования («Возьмите немного больше глазури, чем, как вам кажется, нужно, приняв это во внимание, разровняйте рекурсивно») или сдабривают ее объектно-ориентированной

технологией («но, пожалуйста, придержите объекты анчоусов»). Часто встречается комбинация всех этих стилей.

Но машина регулярных выражений применяет для решения задач совершенно иной подход, являющийся скорее декларативным. Мы описываем цели на языке регулярных выражений, а машина регулярных выражений реализует ту логику, которая требуется для решения наших задач. Логические языки программирования (например, Prolog) не всегда получают такое же освещение, как остальные три стиля, но используются чаще, чем это может показаться. Perl нельзя было бы даже скомпилировать без *make*(1) или *yacc*(1), которые можно рассматривать если не как чисто декларативные языки, то, по крайней мере, как гибриды, в которых соединились императивное и логическое программирование.

В Perl тоже можно поступать подобным образом, смешивая декларации целей с императивным кодом в более значительной степени, чем мы это делали до сих пор, и опираясь на сильные стороны обоих подходов. Можно программным образом образовать строку, которая, в конечном счете, будет представлена машине регулярных выражений, в известном смысле создавая программу, которая пишет новую программу «на лету».

Можно также передавать обычные выражения Perl в качестве заменяющей части оператора `s///` с помощью модификатора `/e`. Это позволяет динамически генерировать строку замены путем выполнения фрагмента кода при обнаружении всякого совпадения шаблона.

Еще более изощренным приемом является вставка с помощью расширения `(?{CODE })` в любое место в середине шаблона фрагментов кода, который будет выполняться каждый раз, когда машина регулярных выражений обнаруживает этот код, двигаясь вперед и назад в этом сложном танце поиска с возвратом.

Наконец, можно применить `s///ee` или `(??{CODE })` для введения еще одного уровня косвенности: сами *результаты* выполнения этих фрагментов кода будут повторно вычисляться и использоваться, создавая фрагменты программы и шаблона на лету в нужный момент.

Генерируемые шаблоны

Уже было сказано,[1] что программы, которые пишут программы, – самые счастливые программы на свете. В книге Джеффри Фридла «Mastering regular expressions» финальная демонстрация изобретательности показывает, как написать программу, создающую регулярное выражение для определения того, удовлетворяет ли строка стандарту RFC 822, т. е. содержится ли в ней совместимый со стандартами допустимый почтовый заголовок. Создаваемый ею шаблон имеет длину в несколько тысяч символов, и читать его примерно так же легко, как двоичный дамп памяти при аварии машины. Однако машина поиска по шаблону Perl этим не озабочена; компиляция

[1] Эндрю Юмом (Andrew Hume), известным философом Unix.

шаблона происходит «без сучка, без задоринки» и, что еще более интересно, поиск выполняется очень быстро – значительно быстрее, чем для многих коротких шаблонов, требующих сложного поиска с возвратом.

Это очень сложный пример. А выше мы привели очень простой пример такой же технологии, когда построили шаблон `$number` из деталей (см. раздел «Интерполяция переменных»). Но чтобы продемонстрировать мощь этого программного подхода при создании шаблона, возьмем задачу средней сложности.

Допустим, необходимо выбрать все слова с определенной последовательностью гласных и согласных; например, «audio» и «eerie» удовлетворяют шаблону `VVCVV` (ГГСГГ)[1]. Хотя нетрудно описать, что считать гласной, а что – согласной, не хочется, чтобы каждый раз нужно было снова это вводить с клавиатуры. Даже в нашем простом случае `VVCVV` нужно ввести шаблон, который выглядит примерно так:

```
^[aeiouy][aeiouy][cbdfghjklmnpqrstvwxzy][aeiouy][aeiouy]$
```

Программа для более широкого применения должна получить строку `VVCVV` и программным образом создать такой шаблон. Еще большая гибкость будет достигнута, если она сможет принять слово, например, `audio`, на входе, использовать его как шаблон для получения `VVCVV`, а из последнего уже получить вышеприведенный шаблон. Кажется сложным, но на практике это не так, поскольку вместо нас генерировать шаблон будет программа. Вот простая программа *cvmap*, которая все это делает:

```
#!/usr/bin/perl
$vowels = 'aeiouy';
$cons   = 'cbdfghjklmnpqrstvwxzy';
%map = (C => $cons, V => $vowels);  # исходное отображение для C (С) и V (Г)

for $class ($vowels, $cons) {       # теперь для каждого типа букв
    for (split //, $class) {        # получить все буквы этого типа
        $map{$_} .= $class;         # и отобразить букву обратно в тип
    }
}

for $char (split //, shift) {       # для каждой буквы в слове шаблона
    $pat .= "[$map{$char}]";        # добавить соответствующий класс символа
}

$re = qr/^${pat}$/i;                # компилировать шаблон
print "REGEX is $re\n";             # отладочная выдача
@ARGV = ('/usr/dict/words')         # выбрать словарь по умолчанию

    if -t && !@ARGV;
while (<>) {                        # пройти через входные данные,
    print if /$re/;                 # выводя все совпадающие строки
}
```

[1] V – первая буква слова «vowel» (гласная), а C – слова «consonant» (согласная). – *Примеч. ред.*

Все интересное содержится в переменной %map. Ключами в ней служат все буквы алфавита, а соответствующими значениями – все буквы их типа. C и V тоже есть в ней, поэтому можно задать либо VVCVV, либо audio, и в любом случае получить eerie. Каждый символ аргумента, переданного программе, используется для извлечения нужного класса символов и добавления его в шаблон. Когда шаблон создан и скомпилирован с помощью qr//, поиск (даже длинного шаблона) происходит быстро. Вот что можно получить, если запустить эту программу с аргументом «fortuitously» (случайно, неожиданно):

```
% cvmap fortuitously /usr/dict/words
REGEX is (?i-xsm:^[cbdfghjklmnpqrstvwxzy][aeiouy][cbdfghjklmnpqrstvwxzy][cbd
fghjklmnpqrstvwxzy][aeiouy][aeiouy][cbdfghjklmnpqrstvwxzy][aeiouy][aeiouy][c
bdfghjklmnpqrstvwxzy][cbdfghjklmnpqrstvwxzy][aeiouycbdfghjklmnpqrstvwxzy]$)
carriageable
circuitously
fortuitously
languorously
marriageable
milquetoasts
sesquiquarta
sesquiquinta
villainously
```

Глядя на это REGEX, можно убедиться в том, какого объема гнусного ввода с клавиатуры удалось избежать благодаря «ленивому» программированию, хотя и окольным путем.

Вычисления при подстановке

Когда модификатор /e («e» от «expression evaluation» – вычисление выражения) используется в выражении s/*PATTERN*/*CODE*/e, заменяющая часть интерпретируется не как строка в двойных кавычках, а как выражение Perl. Это похоже на встроенную команду do { *CODE* }. Несмотря на внешний вид строки это, в действительности, блок кода, компилируемый одновременно с остальной программой задолго до того, как подстановка будет производиться фактически.

Модификатор /e позволяет создавать строки замены более изощренным способом, чем интерполяция в двойных кавычках. Разницу можно показать на примере:

```
s/(\d+)/$1 * 2/;      # Заменяет"42" на "42 * 2"
s/(\d+)/$1 * 2/e;     # Заменяет "42" на "84"
```

А вот как температура по Цельсию переводится в температуру по шкале Фаренгейта:

```
$_ = "Разогреть духовку до 233C.\n";
s/\b(\d+\.?\d*)C\b/int($1 * 1.8 + 32) . "F"/e;   # преобразование в 451F
```

Можно придумать бесконечное число применений этой техники. Вот фильтр, который модифицирует поступающие в него файлы по месту (как редактор), добавляя 100 к каждому числу, которым начинается строчка (и за которым следует двоеточие, на которое мы только смотрим, но не ищем или заменяем):

```
% perl -pi -e 's/^(\d+)(?=:)/100 + $1/e' filename
```

Иногда нужно сделать несколько большее, чем просто использовать найденную строку в другом вычислении. Иногда нужно, чтобы строка и была вычислением, результат которого будет выступать в качестве заменяющего значения. Каждый дополнительный модификатор /e вслед за первым заключает код, который нужно выполнить, в eval. Следующие две строки выполняют одно и то же, но первую из них легче прочесть:

```
s/PATTERN/CODE/ee
s/PATTERN/eval(CODE)/e
```

Этот прием подходит для замены упоминаний простых скалярных переменных на их значения:

```
s/(\$\w+)/$1/eeg;        # Интерполяция большинства значений скаляров
```

Поскольку в действительности используется eval, модификатор /ee находит даже лексические переменные. Слегка более проработанный пример вычисляет замену для простых арифметических выражений с (неотрицательными) целыми числами:

```
$_ = "I have 4 + 19 dollars and 8/2 cents.\n";
s{ (
        \d+ \s*          # найти целое число
        [+*/-]           # и арифметический оператор
        \s* \d+          # и второе целое число
   )
}{ $1 }eegx;             # затем разыменовать $1 и выполнить этот код
print;                   # "I have 23 dollars and 4 cents."
```

Как и для любых других eval *STRING*, перехватываются ошибки этапа компиляции (например, синтаксические) и исключительные ситуации этапа исполнения (например, деление на ноль). Если они возникают, то узнать, что произошло, можно из переменной $@ ($EVAL_ERROR).

Выполнение кода на этапе поиска

В большинстве программ, использующих регулярные выражения, логической последовательностью выполняемых команд руководит управляющая структура этапа исполнения окружающей программы. Мы пишем циклы if или while либо вызываем функции или методы, которые могут привести к вызову операции поиска по шаблону. Даже в s///e управление принадлежит оператору подстановки, а код, находящийся в замене, выполняется только при успехе поиска.

В случае применения *подшаблонов, содержащих код*, обычная связь между регулярным выражением и программным кодом инвертируется. Когда машина регулярных выражений применяет к нашему шаблону свои Правила на этапе поиска, она может наткнуться на расширение регулярного выражения вида `(?{CODE})`. Будучи запущен, этот подшаблон не выполняет поиск или «заглядывание». Это утверждение нулевой ширины, которое всегда «успешно» и выполняется только для получения побочных эффектов. Когда машине регулярных выражений по ходу выполнения шаблона нужно пройти подшаблон, содержащий код, она выполняет этот код.

```
"glyph" =~ /.+ (?{ print "hi" }) ./x;  # Выводит"hi" два раза.
```

Когда машина регулярных выражений пытается сопоставить `glyph` с этим шаблоном, она сначала позволяет `.+` «съесть» все пять букв. Затем выводит `hi`. Когда она находит последнюю точку, все пять букв уже «съедены», поэтому она отступает к `.+` и заставляет его отдать одну из пяти букв. Затем она снова двигается вперед по шаблону, останавливаясь, чтобы еще раз вывести `hi`, перед тем как присвоить `h` конечной точке и успешно завершить поиск.

Фигурные скобки вокруг фрагмента *CODE* имеют целью напомнить, что это блок кода Perl, ведущий себя как блок в лексическом смысле. Это означает, что если в нем для объявления переменной с лексической областью видимости используется `my`, то переменная является закрытой в этом блоке. Но если локализовать переменную с динамической областью видимости при помощи `local`, то результат может оказаться не таким, как мы ожидаем. Подшаблон `(?{CODE })` создает неявную динамическую область видимости, которая действует в оставшейся части шаблона, пока либо поиск не окажется успешным, либо не произойдет возврат через подшаблон с кодом. Можно представлять себе это так, будто возврат из блока по достижении его конца не происходит. Вместо этого осуществляется невидимый рекурсивный вызов машины регулярных выражений для поиска соответствия оставшейся части шаблона. Лишь по завершении этого рекурсивного вызова происходит возврат из блока и снятие локализации с переменных.[1]

В следующем примере мы инициализируем переменную `$i` значением `0`, включив в начало шаблона подшаблон, содержащий код. Затем мы находим

---

[1] Тех, кто знаком с нисходящими рекурсивными анализаторами, такое поведение может смутить, поскольку такие компиляторы возвращаются из вызова рекурсивной функции, как только что-то вычислят. Машина регулярных выражений этого не делает: когда она что-то вычислит, она уходит *глубже* в рекурсию (даже выходя из группы в круглых скобках!). Нисходящий рекурсивный анализатор находится в положении минимальной рекурсии, когда в конце успешно завершает разбор, но наша Машина находится в локальном *максимуме* рекурсии, когда в конце шаблона успешно завершает поиск. Может оказаться полезным мысленно подвесить шаблон за левый край и представить себе его как тощий образ дерева графа вызовов. Если вы можете создать эту картину у себя в голове, то динамическая область видимости локальных переменных становится более понятной. (Если не можете, то хуже вам не станет.)

соответствие любому числу символов с помощью `.*` — но между `.` и `*` мы поместим еще один подшаблон с кодом, чтобы подсчитать, сколько раз `.` находит совпадение.

```
$_ = 'lothlorien';
m/  (?{ $i = 0 })                    # Установить $i в 0
    (.    (?{ $i++ })    )*          # Обновить $i, даже после возврата
    lori                             # Вызывает возврат
 /x;
```

Машина регулярных выражений весело движется вперед, устанавливая `$i` в `0` и позволяя `.*` проглотить все 10 символов строки. Встретив в шаблоне литерал `lori`, она отступает и отдает эти четыре символа из `.*`. После того как найдено соответствие, `$i` по-прежнему будет равна `10`.

Для того чтобы в `$i` находилось число символов, которые в итоге окажутся в `.*`, можно использовать в шаблоне динамическую область видимости:

```
$_ = 'lothlorien';
m/  (?{ $i = 0 })
    (. (?{ local $i = $i + 1; }) )*  # Обновление $i,не зависящее от возвратов.
    lori
    (?{ $result = $i })              # Копировать по нелокализованному адресу.
 /x;
```

Здесь посредством `local` мы гарантируем нахождение в `$i` числа символов, найденных `.*`, независимо от возвратов при поиске. `$i` будет забыта, когда закончится регулярное выражение, поэтому подшаблон с кодом `(?{ $result = = $i })` сохраняет этот счетчик в `$result`.

Специальная переменная `$^R` (описанная в главе 28) содержит результат последнего `(?{`*CODE*`})`, выполненного в составе успешного поиска.

Расширение `(?{`*CODE*`})` можно использовать в качестве условия *COND* в `(?(`*COND*`)` *IFTRUE*`|`*IFFALSE*`)`. При этом `$^R` не устанавливается, а круглые скобки вокруг условного оператора можно опустить:

```
"glyph" =~ /.+(?(?{ $foo{bar} gt "symbol" }).|signet)./;
```

Здесь мы проверяем, больше ли `$foo{bar}`, чем `symbol`. Если да, то в шаблон включается `.`, а если нет, то `signet`. Несколько растянув эту команду, можно сделать ее более удобочитаемой:

```
"glyph" =~ m{
    .+                            # несколько чего угодно
    (?(?{                         # если
            $foo{bar} gt "symbol"  # это правда
        })
            .                     # найти еще что угодно
        |                         # иначе
            signet                # найти signet
```

```
    )
    .                                   # и еще что угодно
}x;
```

Когда действует `use re 'eval'`, регулярное выражение может содержать подшаблоны `(?{CODE})`, даже если в нем интерполируются переменные:

```
/(.*?) (?{length($1) < 3 && warn}) $suffix/;  # Ошибка без use re 'eval'
```

Обычно это запрещено, поскольку представляет собой угрозу безопасности. Даже если приведенный шаблон невинен, поскольку невинен `$suffix`, анализатор регулярных выражений не может различить, какие части строки интерполировались, а какие нет, поэтому он просто запрещает все подшаблоны с кодом, если производилась интерполяция.

Если шаблон получен из меченых данных, то даже `use re 'eval'` не разрешит поиск по шаблону.

Когда действует `use re 'taint'` и к меченой строке применяется регулярное выражение, захваченные подшаблоны (в нумерованных переменных или списке значений, возвращаемом `m//` в списковом контексте) помечаются. Это удобно, если операции регулярного выражения над мечеными данными имеют целью не извлечение безопасных подстрок, а просто осуществление других преобразований. Дополнительные сведения о меченых данных см. в главе 23 «Защита данных». Для этой прагмы прекомпилированные регулярные выражения (обычно получаемые через `qr//`) не считаются интерполированными:

```
/foo${pat}bar/
```

Это допустимо, если `$pat` представляет собой прекомпилированное регулярное выражение, даже если `$pat` содержит подшаблоны с кодом `(?{CODE})`.

Выше мы привели небольшой фрагмент вывода `use re 'debug'`. Более упрощенное решение для отладки состоит в использовании подшаблонов с кодом `(?{CODE})` для вывода соответствия, уже найденного в процессе поиска:

```
"abcdef" =~ / .+ (?{print "Пока найдено: $&\n"}) bcdef $/x;
```

В результате выводится:

```
Пока найдено: abcdef
Пока найдено: abcde
Пока найдено: abcd
Пока найдено: abc
Пока найдено: ab
Пока найдено: a
```

что показывает, как `.+` захватил все буквы, и отдает их по одной по мере возврата машины регулярных выражений.

Интерполяция шаблона на этапе поиска

Можно строить части шаблона, находясь в нем самом. Расширение (??{*CODE* }) делает возможным добавление кода, результатом выполнения которого является допустимый шаблон. Похоже на то, чтобы сказать /$pattern/, за исключением того, что $pattern можно генерировать на этапе исполнения или, точнее, на этапе поиска. Например:

```
/\w (??{ if ($threshold > 1) { "red" } else { "blue" } }) \d/x;
```

Это эквивалентно /\wred\d/, если $threshold больше 1, и /\wblue\d/ в противном случае.

В исполняемый код можно включать ссылки на найденный текст из только что найденных подстрок (даже если позже они окажутся ненайденными из-за возврата). Например, следующая команда находит все строки, которые одинаково читаются в прямом и обратном направлении (известные как палиндромы, фразы «с горбом посередине»):

```
/^ (.+) .? (??{quotemeta reverse $1}) $/xi;
```

Нейтрализовать круглые скобки можно так:

```
$text =~ /( \(+ ) (.*?) (??{ '\)' x length $1 })/x;
```

Это соответствует строкам вида (shazam!) и (((shazam!))), помещая shazam! в $2. К несчастью, этот шаблон не обнаруживает, есть ли парные скобки в середине. Для этого требуется рекурсия.

Но очень хорошо, что можно создавать и рекурсивные шаблоны. Можно заставить скомпилированный шаблон, использующий (??{*CODE* }), ссылаться на себя самого. Рекурсивный поиск оказывается довольно нерегулярным. В любой книге по регулярным выражениям указывается, что стандартные регулярные выражения не могут правильно находить вложенные скобки. И это правильно. Правильно и то, что регулярные выражения в Perl не являются стандартными. Следующий шаблон[1] находит набор вложенных скобок, сколь бы глубоко они ни были вложены:

```
$np = qr{
           \(
           (?:
              (?> [^()]+ )    # Нескобки без возврата
            |
              (??{ $np })     # Группировать с найденными скобками
           )*
           \)
        }x;
```

[1] Обратите внимание, что нельзя объявить переменную в той же команде, в которой мы собираемся ее использовать. Разумеется, всегда можно объявить ее раньше.

Вот как можно с его помощью организовать поиск вызова функции:

```
$funpat = qr/\w+$np/;
'myfunfun(1,(2*(3+4)),5)' =~ /^$funpat$/;   # Соответствует!
```

Условная интерполяция

Расширение регулярных выражений (?(*COND*) *IFTRUE*|*IFFALSE*) аналогично оператору ?: в Perl. Если *COND* истинно, применяется шаблон *IFTRUE*; в противном случае – шаблон *IFFALSE*. Условие *COND* может быть ссылкой на найденный текст (выражаемой как «голое» целое число без \ или $), утверждением с заглядыванием или подшаблоном с кодом. (См. разделы «Утверждения с заглядыванием» и «Выполнение кода на этапе поиска» выше в этой главе.)

Если *COND* представляет собой целое число, оно рассматривается как ссылка на найденный текст. Рассмотрим пример:

```
#!/usr/bin/perl

$x = 'Perl является бесплатным.';
$y = 'ManagerWare стоит $99.95.';

foreach ($x, $y) {
    /^(\w+) (?:является|(стоит)) (?(2)(\$\d+)|\w+)/;  # Либо (\$\d+), либо \w+
    if ($3) {
        print "$1 стоит денег.\n";                  # ManagerWare стоит денег.
    } else {
        print "$1 распространяется бесплатно.\n";  # Perl распространяется
                                                     бесплатно.
    }
}
```

Здесь *COND* есть (2), что истинно, когда существует вторая ссылка на найденный текст. Если это так, то (\$\d+) включается в шаблон в этой точке (создавая ссылку на найденный текст $3); в противном случае используется \w+.

Если *COND* представляет собой утверждение с заглядыванием или подшаблоном, содержащим код, то определить, надо ли включать *IFTRUE* или *IFFALSE*, можно на основании истинности утверждения:

```
/[ATGC]+(?(?<=AA)G|C)$/;
```

Здесь в качестве *COND* применяется утверждение с заглядыванием назад, чтобы найти в ДНК последовательность, которая оканчивается на AAG или другую основную комбинацию и C.

Альтернатива |*IFFALSE* может быть опущена. В этом случае шаблон *IFTRUE* будет включен в шаблон, как обычно, если истинно *COND*, но если условие не истинно, машина регулярных выражений переместится к следующей части шаблона.

Определение собственных утверждений

Нельзя изменить работу машины регулярных выражений в Perl, но те, кто уже достаточно начитался о сложных шаблонах, могут попробовать изменить то, как она видит шаблон. Поскольку Perl интерпретирует шаблон аналогично строке в двойных кавычках, то можно использовать чудеса перегруженных строковых констант для того, чтобы избранные нами текстовые последовательности автоматически транслировались в другие.

В приведенном ниже примере мы задаем два преобразования, которые должны происходить, когда Perl обнаруживает шаблон. Во-первых, мы определим `\tag` таким образом, что когда он встречается в шаблоне, то автоматически транслируется в `(?:<.*?>)`, что соответствует большинству тегов HTML и XML. Во-вторых, мы «переопределим» метазнак `\w` так, чтобы он работал только с буквами английского алфавита.

Мы определим пакет с именем `Tagger`, который укроет перегрузку от нашей основной программы. Сделав это, мы сможем сказать:

```
use Tagger;
$_ = '<I>верблюд</I>';
print "Найден верблюд в тегах" if /\tag\w+\tag/;
```

Вот *Tagger.pm*, оформленный в виде модуля Perl (см. главу 11):

```
package Tagger;
use overload;

sub import { overload::constant 'qr' => \&convert }

sub convert {
    my $re = shift;
    $re =~ s/ \\tag  /<.*?>/xg;
    $re =~ s/ \\w    /[A-Za-z]/xg;
    return $re;
}
1;
```

Модуль `Tagger` получает шаблон непосредственно перед интерполяцией, поэтому обойти перегрузку можно, обойдя интерполяцию, например:

```
$re = '\tag\w+\tag';   # Эта строка начинается с \t, символа табуляции
print if /$re/;        # Ищет символ табуляции, за которым следует "a"...
```

Для того чтобы преобразовать интерполированную переменную, вызовите функцию `convert` непосредственно:

```
$re = '\tag\w+\tag';          # Эта строка начинается с \t, символа табуляции
$re = Tagger::convert $re;    # разыменовать \tag и \w
print if /$re/;               # $re становится <.*?>[A-Za-z]+<.*?>
```

Если вы все еще не поняли, что это за штуки `sub` в модуле `Tagger`, то скоро поймете, потому что этому посвящена вся следующая глава.

6

Подпрограммы

Как и многие другие языки, Perl предоставляет возможность создания подпрограмм, определяемых пользователем.[1] Эти подпрограммы могут быть определены в любом месте основной программы, загружены из других файлов с помощью ключевых слов `do`, `require` или `use` либо сгенерированы на этапе исполнения с помощью `eval`. Можно даже загружать их на этапе исполнения с помощью механизма, описываемого в разделе «Автозагрузка» в главе 10 «Пакеты». Вызов подпрограмм может осуществляться косвенным образом посредством переменной, содержащей имя программы или ссылку на нее, либо через объект, позволяя последнему определить, какая подпрограмма в действительности должна быть вызвана. Можно создавать анонимные подпрограммы, доступ к которым осуществляется только через ссылки, и при желании использовать их для клонирования новых почти идентичных функций через *замыкания* (*closures*), описываемые в одноименном разделе главы 8 «Ссылки».

Синтаксис

Для того чтобы объявить именованную подпрограмму, не определяя ее, используется один из следующих форматов:

---

[1] Мы называем их также *функциями*, но в Perl функции и подпрограммы – одно и то же. Иногда мы будем даже называть их *методами*, которые определяются так же, как подпрограммы, но вызываются иным способом.

```
sub NAME
sub NAME PROTO
sub NAME       ATTRS
sub NAME PROTO ATTRS
```

Именованную подпрограмму можно объявить и определить, добавив блок *BLOCK*:

```
sub NAME              BLOCK
sub NAME PROTO        BLOCK
sub NAME       ATTRS  BLOCK
sub NAME PROTO ATTRS  BLOCK
```

Для создания анонимной подпрограммы или замыкания надо опустить *NAME*:

```
sub                   BLOCK
sub      PROTO        BLOCK
sub             ATTRS BLOCK
sub      PROTO ATTRS  BLOCK
```

PROTO и *ATTRS* обозначают прототип и атрибуты; о том и другом будет рассказано в соответствующих разделах далее в этой главе. *PROTO* и *ATTRS* не столь важны: существенными частями являются *NAME* и *BLOCK*, даже если они отсутствуют.

Для форматов с отсутствующим *NAME* должен все же иметься какой-то способ вызова подпрограммы. Для этого нужно сохранить возвращаемое значение, потому что эта форма объявления sub не только компилируется на этапе компиляции, как можно было предполагать, но и создает возвращаемое значение этапа исполнения:

```
$subref = sub BLOCK;
```

Для того чтобы импортировать подпрограммы, определенные в другом модуле, нужно сказать:

```
use MODULE qw(NAME1 NAME2 NAME3...);
```

Прямой вызов подпрограмм осуществляется так:

```
NAME(LIST)          # & не обязателен, если есть скобки.
NAME LIST           # Скобки не обязательны, если sub имеет объявление/
                      импортирована.
&NAME               # Передает подпрограмме текущее значение @_
                    # (и обходит прототип).
```

Для косвенного вызова подпрограмм (по имени или по ссылке) используется синтаксис:

```
&$subref(LIST)      # & нельзя опускать при косвенном вызове
$subref->(LIST)     # (если только не используется инфиксная запись).
&$subref            # Передает подпрограмме текущее значение @_.
```

Официальное имя подпрограммы содержит префикс &. Можно вызывать подпрограмму с указанием префикса, но обычно он не является обязательным, как и круглые скобки, если подпрограмма предварительно объявлена. Однако & не является необязательным, когда подпрограмме нужно просто указать имя, как при использовании в качестве аргумента функций defined или undef, либо при создании ссылки на именованную подпрограмму, как в $subref = \&name. Нельзя опускать & и в тех случаях, когда осуществляется косвенный вызов подпрограммы в конструкциях &$subref() или &{$subref}(). Однако в более удобном обозначении $subref->() амперсанд не требуется. Дополнительные сведения о ссылках на подпрограммы см. в главе 8 «Ссылки».

Perl не требует какого-либо особого стиля заглавных букв в именах подпрограмм. Однако одно нестрого соблюдаемое соглашение состоит в том, что имена функций, неявно вызываемых системой этапа исполнения Perl (BEGIN, CHECK, INIT, END, AUTOLOAD, DESTROY и всех функций, указанных в главе 14 «Связанные переменные»), пишутся заглавными буквами, поэтому такого стиля, возможно, следует избегать. (Но подпрограммы, используемые для константных величин, обычно тоже пишутся заглавными буквами. Это нормально. Надеемся...)

Семантика

Пока вы не слишком устали от всего этого синтаксиса, просто запомните, что обычный способ определения простой подпрограммы выглядит примерно так:

```
sub razzle {
    print "Ok, you've been razzled.\n";
}
```

а обычный способ ее вызова просто:

```
razzle();
```

В данном случае мы игнорировали входные данные (аргументы) и выходные (возвращаемые значения). Но действующая в Perl модель передачи данных в подпрограмму и из нее на самом деле очень проста: все параметры функции передаются как один плоский список скаляров; аналогично, множественные возвращаемые значения возвращаются в место вызова как один плоский список скаляров. Как и для любого списка, все массивы и хеши, передаваемые в этих списках, интерполируют свои значения в плоский список, утрачивая свою индивидуальность, но есть несколько способов обойти это, и часто весьма полезной оказывается автоматическая интерполяция списка. Как список параметров, так и список возвращаемых значений могут содержать сколь угодно много или сколь угодно мало скалярных элементов (хотя можно наложить ограничения на список параметров путем использования прототипов). В действительности архитектура Perl основывается на понятии *вариадических* (*variadic*) функций (таких, которые принимают любое

число аргументов), в отличие от C, куда они довольно неуклюже втиснуты, чтобы можно было вызывать *printf*(3).

Итак, собираясь создавать язык на основе идеи передачи переменного числа произвольных аргументов, следует подумать о том, как облегчить обработку этих произвольных списков аргументов. Любые аргументы, передаваемые в программу Perl, поступают в нее в виде массива `@_`. Если функция вызывается с двумя аргументами, то внутри функции они доступны как первые два элемента этого массива: `$_[0]` и `$_[1]`. Поскольку `@_` представляет собой обычный массив с необычным именем, с ним можно делать все то же, что обычно делают с массивом.[1] Массив `@_` является локальным, но его значения служат псевдонимами для реальных скалярных параметров. (Это называется семантикой передачи по ссылке.) Поэтому можно изменять фактические параметры, изменяя соответствующие элементы `@_`. (Однако это делается редко, потому что в Perl очень легко возвращать значения, которые представляют интерес.)

Значение, возвращаемое подпрограммой (как и любым другим блоком), есть значение выражения, вычисленного последним. Либо можно использовать явно команду `return`, чтобы задать возвращаемое значение и выйти из подпрограммы в любой ее точке. В любом случае, если подпрограмма вызывается в скалярном или списковом контексте, то и конечное выражение в подпрограмме вычисляется в том же самом скалярном или списковом контексте.

Приемы работы со списками параметров

В Perl нет именованных формальных параметров, но на практике нужно лишь скопировать значения `@_` в список `my`, который прекрасно служит в качестве списка формальных параметров. (Не случайно, что копирование значений заменяет семантику передачи по ссылке семантикой передачи по значению, на которую обычно и рассчитывают при работе с параметрами даже те, кто не знает специальных терминов, используемых для этого в информатике.) Вот типичный пример:

```
sub maysetenv {
    my ($key, $value) = @_;
    $ENV{$key} = $value unless $ENV{$key};
}
```

Но от вас не требуется давать имена параметрам, в чем состоит весь смысл массива `@_`. Например, для определения максимума можно непосредственно пройти по массиву `@_`:

```
sub max {
    my $max = shift(@_);
    for my $item (@_) {
```

[1] Это область, в которой Perl *более* ортогонален, чем типичные языки программирования.

```
        $max = $item if $max < $item;
    }
    return $max;
}

$bestday = max($mon,$tue,$wed,$thu,$fri);
```

Либо можно заполнить сразу весь хеш:

```
sub configuration {
    my %options = @_;
    print "Максимально подробно.\n" if $options{VERBOSE} == 9;
}
configuration(PASSWORD => "xyzzy", VERBOSE => 9, SCORE => 0);
```

Вот пример, показывающий, как, не обращаясь к формальным аргументам, модифицировать фактические:

```
upcase_in($v1, $v2);  # эта подпрограмма изменяет $v1 и $v2
sub upcase_in {
    for (@_) { tr/a-z/A-Z/ }
}
```

Конечно, таким способом нельзя модифицировать константы. Если бы аргумент оказался скалярным литералом, например «хоббит», или скалярной переменной, доступной только для чтения, например `$1`, и вы попытались бы его изменить, Perl возбудил бы исключительную ситуацию (предположительно, фатальную; возможно, угрожающую вашей карьере). Например, следующий вызов не будет работать:

```
upcase_in("фредерик");
```

Было бы значительно безопаснее написать функцию `upcase_in` так, чтобы она возвращала копию своих параметров, а не изменяла их по месту:

```
($v3, $v4) = upcase($v1, $v2);
sub upcase {
    my @parms = @_;
    for (@parms) { tr/a-z/A-Z/ }
    # Проверить, осуществлен ли вызов в списковом контексте.
    return wantarray ? @parms : $parms[0];
}
```

Обратите внимание, что этой функции (без прототипа) безразлично, что ей передается: скаляры или массивы. Perl все расплющит в один большой, длинный, плоский список параметров `@_`. Это один из случаев, когда проявляется простота стиля передачи аргументов в Perl. Функция `upcase` будет отлично работать без изменения определения `upcase`, даже если передавать ей такие аргументы:

```
@newlist = upcase(@list1, @list2);
@newlist = upcase( split /:/, $var );
```

Не соблазнитесь, однако, написать такое:

```
(@a, @b) = upcase(@list1, @list2);   # НЕВЕРНО
```

Почему так нельзя? Потому что, как и список входных параметров @_, список возвращаемых значений является плоским. В результате он запишет все свои значения в @a и очистит @b, записав туда нулевой список. Как этого избежать, см. далее в разделе «Передача ссылок».

Индикация ошибок

Если вы хотите организовать возврат из функции так, чтобы в месте вызова можно было определить, что произошла ошибка, то проще всего сделать это в Perl с помощью простой команды return без аргумента. В результате при вызове функции в скалярном контексте возвращается undef, а в списковом – нулевой список.

В крайних обстоятельствах для индикации ошибки может оказаться предпочтительным возбуждение исключительной ситуации. Однако пользуйтесь этим средством экономно, иначе вся программа окажется замусоренной обработчиками исключительных ситуаций. Например, неудача при открытии файла в общей функции открытия файлов едва ли является исключительным событием. Однако игнорирование ее вполне может быть таковым. Встроенная функция wantarray возвращает undef, если ваша функция вызвана в пустом контексте, поэтому вы можете узнать, что вас игнорируют:

```
if ($something_went_awry) {
    return if defined wantarray;  # хорошо, не пустой контекст.
    die "Да ты ??смотри на мою ошибку, сокет висячий!!!\n";
}
```

Вопросы областей видимости

Подпрограммы могут вызываться рекурсивно, потому что каждый вызов получает собственный массив аргументов, даже если программа вызывает саму себя. Если подпрограмма вызывается с использованием формата &, то список аргументов не обязателен. Если применяется &, но список аргументов опущен, происходит нечто особое: неявным образом передается массив @_ вызывающей процедуры. Это механизм повышения эффективности, к которому новичкам, возможно, не стоит обращаться.

```
&foo(1,2,3);    # передача трех аргументов
foo(1,2,3);     # то же самое

foo();          # передача пустого списка
&foo();         # то же самое

&foo;           # foo() получает текущие аргументы, как foo(@_), но быстрее!
foo;            # то же, что foo(), если было объявление sub foo, иначе голое
                  слово "foo"
```

Формат & не только делает необязательным список аргументов, но и отключает проверку прототипа для передаваемых аргументов. Отчасти это обусловлено историческими причинами, а отчасти сделано для того, чтобы было удобнее мошенничать, если вы хорошо понимаете, что делаете. См. раздел «Прототипы» далее в этой главе.

Переменные, к которым вы обращаетесь в функции, не определенные в ней как закрытые, не обязательно будут глобальными переменными: они подчиняются обычным правилам Perl для областей видимости блоков. Как объяснялось в разделе «Имена» главы 2 «Всякая всячина», это значит, что попытка разрешения их сначала предпринимается в окружающей лексической области (или областях) видимости, а затем в области видимости пакета. Поэтому с точки зрения подпрограммы все переменные my из охватывающей области видимости вполне доступны.

Например, приведенная ниже функция bumpx имеет доступ к лексической переменной $x с областью видимости в файле, поскольку область видимости, в которой было объявление my – сам файл – не была закрыта раньше, чем определена подпрограмма:

```
# начало файла
my $x = 10;          # объявление и инициализация переменной
sub bumpx { $x++ }   # функция может видеть внешнюю лексическую переменную
```

Программисты на C и C++ представят себе, вероятно, $x как «файловую статическую» переменную. Она закрыта для функций в других файлах, но глобальна с точки зрения функций, определенных после объявления my. Программисты на C, которые ищут в Perl то, что они назвали бы «статическими переменными» для файлов или функций, не находят в нем ключевого слова «static». Программисты на Perl обычно избегают этого слова, потому что статические системы мертвы и скучны, а история употребления этого слова слишком запутана.

Хотя в лексиконе Perl нет слова «static», программисты Perl не испытывают затруднений при создании переменных, являющихся закрытыми в функции и сохраняющими свое значение между вызовами функции. Просто для этого нет специального слова. Более богатые примитивы областей видимости Perl сочетаются с автоматическим управлением памятью так, как и мечтать не могли те, кто ищет ключевое слово «static».

Лексические переменные не исчезают при автоматической сборке мусора, когда происходит выход из их области видимости; они попадают в переработку только тогда, когда больше не *используются*, что значительно важнее. Чтобы создать закрытые переменные, которые не сбрасываются автоматически между обращениями к функции, заключите всю функцию в дополнительный блок, в который поместите также объявление my и определение функции. Можно помещать в блок более одной функции, и тогда они получают совместный доступ к переменной, которая иначе была бы закрытой:

```
{
    my $counter = 0;
```

```
    sub next_counter { return ++$counter }
    sub prev_counter { return --$counter }
}
```

Как всегда, доступ к лексической переменной ограничен кодом в той же области лексической видимости. С другой стороны, имена двух функций доступны глобально (внутри пакета) и, поскольку они определены внутри области видимости $counter, то могут обращаться к этой переменной, хотя никто больше делать этого не может.

Если функция загружена через require или use, то все может быть прекрасно. Если все происходит в основной программе, то нужно обеспечить, чтобы на этапе исполнения какое-либо присвоение переменной my выполнялось достаточно рано. Для этого можно поместить весь блок перед основной программой, либо поместить все в блок BEGIN или INIT, чтобы гарантировать выполнение до запуска программы:

```
BEGIN {
    my @scale = ('A' .. 'G');
    my $note  = -1;
    sub next_pitch { return $scale[ ($note += 1) %= @scale ] };
}
```

Блок BEGIN не оказывает воздействия на определение подпрограммы и на сохранение значений лексическими переменными, используемыми подпрограммой. Он нужен только для того, чтобы гарантировать инициализацию переменных до первого вызова подпрограммы. Дополнительные сведения об объявлении закрытых и глобальных переменных можно найти в описаниях my и our, соответственно, в главе 29 «Функции». Конструкции BEGIN и INIT описываются в главе 8 «Компиляция».

Передача ссылок

Если требуется передать более одного массива в функцию или из нее, и вы хотите поддержать их целостность, нужно использовать механизм явной передачи по ссылке. Но раньше необходимо разобраться в деталях действия ссылок, описанных в главе 8 «Ссылки». В противном случае эта глава может оказаться не понятой вами. Однако вы всегда можете посмотреть картинки...

Вот несколько простых примеров. Во-первых, определим функцию, которая предполагает получение ссылки на массив. Если массив большой, то значительно быстрее передать его как одну ссылку, а не длинный список значений:

```
$total = sum ( \@a );

sub sum {
    my ($aref)  = @_;
    my ($total) = 0;
    foreach (@$aref) { $total += $_ }
```

```
    return $total;
}
```

Передадим в функцию несколько массивов и заставим ее выталкивать их все из стека, возвращая новый список, состоящий из их последних элементов:

```
@tailings = popmany ( \@a, \@b, \@c, \@d );

sub popmany {
    my @retlist = ();
    for my $aref (@_) {
        push @retlist, pop @$aref;
    }
    return @retlist;
}
```

Вот как можно написать функцию, которая находит пересечение множеств, возвращая список ключей, имеющихся в каждом из переданных ей хешей:

```
@common = inter( \%foo, \%bar, \%joe );

sub inter {
    my %seen;
    for my $href (@_) {
        while (my $k = each %$href ) {
            $seen{$k}++;
        }
    }
    return grep { $seen{$_} == @_ } keys %seen;
}
```

Пока мы задействовали обычный механизм возврата списка. Что происходит, если требуется передать или возвратить хеш? Если хеш только один или вы не возражаете против того, чтобы они объединились в один, то сойдет и обычное соглашение по вызовам, хотя обходится оно дороговато.

Как мы уже отмечали, беда случается при такой записи:

```
(@a, @b) = func(@c, @d);
```

или такой:

```
(%a, %b) = func(%c, %d);
```

Этот синтаксис не работает. Он просто устанавливает `@a` или `%a` и очищает `@b` или `%b`. Кроме того, эта функция не получает в качестве аргументов два отдельных массива или хеша: она получает, как обычно, один длинный список в `@_`.

Допустим, требуется, чтобы функции использовали ссылки как для входных, так и для выходных данных. Вот функция, которая принимает в качестве аргументов две ссылки на массив и возвращает две ссылки на массив, упорядоченные по числу элементов в этих массивах:

```
($aref, $bref) = func(\@c, \@d);
print "@$aref больше, чем @$bref\n";
sub func {
    my ($cref, $dref) = @_;
    if (@$cref > @$dref) {
        return ($cref, $dref);
    } else {
        return ($dref, $cref);
    }
}
```

О том, как передавать в функции и из них дескрипторы файлов или каталогов, см. разделы «Ссылки на дескрипторы файлов» и «Ссылки на таблицы символов» главы 8.

Прототипы

Perl позволяет пользователю определять свои функции так, чтобы они вызывались как встроенные функции Perl. Рассмотрим `push(@array, $item)`, в которой подразумевается, что она получит ссылку на `@array`, а не список значений, содержащихся в `@array`, чтобы можно было модифицировать массив. *Прототипы* дают возможность определять подпрограммы так, чтобы они принимали аргументы как многие встроенные функции, т. е. с определенными ограничениями на число и типы аргументов. Мы называем их «прототипы», но они действуют, скорее, как автоматические шаблоны для контекста вызова, чем как то, что называют прототипами программисты на C или Java. С помощью этих шаблонов Perl автоматически добавляет неявные символы обратной косой черты или вызовы `scalar` и все прочее, благодаря чему полученный результат должен выглядеть согласно шаблону. Например, если объявить:

```
sub mypush (\@@);
```

то `mypush` принимает аргументы точно так, как это делает `push`. Для того чтобы реализовать это, объявление функции должно быть видимо на этапе компиляции. Прототип влияет на интерпретацию вызовов функции только тогда, когда символ `&` опущен. Иными словами, если вызвать ее как встроенную функцию, она и ведет себя как встроенная функция. А если вызвать ее как обычную подпрограмму, она и ведет себя как обычная подпрограмма. Символ `&` подавляет проверку прототипа и соответствующие эффекты контекста.

Поскольку прототипы принимаются во внимание только на этапе компиляции, естественно, следует, что они не оказывают влияния на ссылки на подпрограммы типа `\&foo` или косвенный вызов подпрограмм типа `&{$subref}` или `$subref->()`. На вызовы методов прототипы также не оказывают влияния. Дело в том, что фактически вызываемая функция не определена на этапе компиляции и зависит от наследования, которое определяется в Perl динамически.

Основное назначение прототипов в том, чтобы дать программисту возможность определять подпрограммы, действующие как встроенные функции. Вот несколько прототипов, которые можно использовать для эмуляции соответствующих встроенных функций:

Объявление	Вызов
`sub mylink ($$)`	`mylink $old, $new`
`sub myreverse (@)`	`myreverse $a,$b,$c`
`sub myjoin ($@)`	`myjoin ":",$a,$b,$c`
`sub mypop (\@)`	`mypop @array`
`sub mysplice (\@$$@)`	`mysplice @array,@array,0,@pushme`
`sub mykeys (\%)`	`mykeys %{$hashref}`
`sub mypipe (**)`	`mypipe READHANDLE, WRITEHANDLE`
`sub myindex ($$;$)`	`myindex &getstring, "substr"` `myindex &getstring, "substr", $start`
`sub mysyswrite (*$;$$)`	`mysyswrite OUTF, $buf` `mysyswrite OUTF, $buf, length($buf)-$off, $off`

`sub myopen (*;$@)`	`myopen HANDLE`
	`myopen HANDLE, $name`
	`myopen HANDLE, "-|", @cmd`

`sub mygrep (&@)`	`mygrep { /foo/ } $a,$b,$c`
`sub myrand ($)`	`myrand 42`
`sub mytime ()`	`mytime`

Каждый символ прототипа с обратной косой чертой перед ним (в скобках в левой колонке) представляет фактический аргумент (пример которого приведен в правой колонке), который обязательно должен начинаться с этого символа. Как первый аргумент `keys` должен начинаться с `%`, так и первый аргумент `mykeys` должен начинаться с `%`.

Точка с запятой отделяет обязательные аргументы от необязательных. (Перед `@` или `%` ее употребление излишне, т. к. списки могут быть нулевыми.) Символы прототипа без обратной косой черты имеют особый смысл. Каждый символ `@` или `%` без обратной косой черты съедает все остальные фактические аргументы и устанавливает списковый контекст. (Это эквивалентно *`LIST`* в описании синтаксиса.) Для аргумента, представленного `$`, устанавливается списковый контекст. Символ `&` требует ссылки на именованную или анонимную подпрограмму.

Символ `*` позволяет подпрограмме принимать в этом месте все, что было бы принято встроенной функцией как дескриптор файла: голое имя, константа, скалярное выражение, typeglob или ссылка на typeglob. Это значение будет доступно в подпрограмме как простой скаляр или (в двух последних слу-

чаях) как ссылка на typeglob. Для того чтобы такие аргументы всегда преобразовывались в ссылки на typeglob, используйте `Symbol::qualify_to_ref` следующим образом:

```
use Symbol 'qualify_to_ref';

sub foo (*) {
    my $fh = qualify_to_ref(shift, caller);
    ...
}
```

Обратите внимание, что последние три примера в вышеприведенной таблице анализатор обрабатывает особым образом. `mygrep` анализируется как настоящий списковый оператор, `myrand` — как настоящий унарный оператор с тем же унарным приоритетом, что и у `rand`, а `mytime` — как настоящая функция без аргументов, типа `time`.

Это значит, что, сказав:

```
mytime +2;
```

вы получите `mytime() + 2`, а не `mytime(2)`, как было бы, если бы она анализировалась без прототипа или с унарным прототипом.

Пример `mygrep` показывает также, что `&` обрабатывается особым образом, когда является первым аргументом. Обычно прототип с `&` требует аргумента типа `\&foo` или `sub{}`. Однако, если это первый аргумент, можно пропустить `sub` в анонимной подпрограмме и просто передать голый блок на место «косвенного объекта» (без последующей запятой). Поэтому изящество прототипа `&` состоит в том, что с его помощью можно создавать новый синтаксис при условии, что `&` находится в первой позиции:

```
sub try (&$) {
    my ($try, $catch) = @_;
    eval { &$try };
    if ($@) {
        local $_ = $@;
        &$catch;
    }
}
sub catch (&) { $_[0] }
try {
    die "phooey";
}                   # не конец вызова функции!
catch {
    /phooey/ and print "unphooey\n";
};
```

В результате выводится `unphooey`. В этом примере `try` вызывается с двумя аргументами: анонимной функцией `{die "phooey";}` и значением, возвращаемым функцией `catch`, которое в данном случае является ее первым аргументом — всем блоком другой анонимной функции. В `try` вызывается первая

функция-аргумент, являющаяся защищенной в блоке eval, который перехватит ошибку, если она возникнет. Если это произойдет, вызывается вторая функция с локализованным вариантом глобальной переменной $_, которая установлена в соответствии с возбужденной исключительной ситуацией.[1] Если все это выглядит для вас полной неразберихой, почитайте о функциях die и eval в главе 29, а затем об анонимных функциях и замыканиях в главе 8. С другой стороны, те, кого это заинтересовало, могли бы изучить модуль Error из CPAN, в котором такая технология применяется для реализации тщательно структурированной обработки исключительных ситуаций с использованием предложений try, catch, except, otherwise и finally.

Вот новая реализация оператора grep (встроенная функция, конечно, более эффективна):

```
sub mygrep (&@) {
    my $coderef = shift;
    my @result;
    foreach $_ (@_) {
        push(@result, $_) if &$coderef;
    }
    return @result;
}
```

Кое-кто предпочел бы, чтобы имелись полные буквенно-цифровые прототипы. Буквы и цифры намеренно выведены из прототипов с явной целью когда-нибудь ввести именованные формальные параметры. (Может быть.) Основная задача имеющегося в настоящее время механизма состоит в том, чтобы позволить тем, кто модули пишет, осуществлять в некотором объеме на этапе компиляции контроль над теми, кто модулями пользуется.

Подставляемые функции-константы

Функции, прототип которых содержит пустые скобки (), и, следовательно, не принимающие никаких аргументов, анализируются аналогично встроенной time. Что интересно, компилятор рассматривает такие функции как кандидаты на подстановку (inlining). Если результатом такой функции, после оптимизации и свертывания констант, становится константа или скаляр с лексической областью видимости без других ссылок, то это значение будет использоваться вместо вызова данной функции. Однако вызовы вида *&NAME* никогда не преобразуются в подстановку, так же, как не подвергаются другим воздействиям прототипов. (См. в главе 31 «Модули прагм» описание директивы use constant, где рассказывается, как объявлять такие константы.)

---

[1] Да, остались неразрешенные проблемы, связанные с областью видимости @_. Этот вопрос мы в данное время игнорируем. Но если мы когда-нибудь установим для @_ лексическую область видимости, как это уже делается в экспериментальных многопоточных версиях Perl, то эти анонимные подпрограммы смогут действовать как замыкания.

Обе следующие версии функций для вычисления числа π компилятор сделает подставляемыми:

```
sub pi ()           { 3.14159 }             # Не точно, но близко
sub PI ()           { 4 * atan2(1, 1) }     # Точно, насколько можно
```

На практике все следующие функции будут подставляемыми, поскольку Perl сможет все определить на этапе компиляции:

```
sub FLAG_FOO ()     { 1 << 8 }
sub FLAG_BAR ()     { 1 << 9 }
sub FLAG_MASK ()    { FLAG_FOO | FLAG_BAR }

sub OPT_GLARCH ()   { (0x1B58 & FLAG_MASK) == 0 }
sub GLARCH_VAL ()   {
    if (OPT_GLARCH) { return 23 }
    else            { return 42 }
}

sub N () { int(GLARCH_VAL) / 3 }
BEGIN {                 # компилятор выполняет этот блок на этапе компиляции
    my $prod = 1;       # сохраняющая значение закрытая переменная
    for (1 .. N) { $prod *= $_ }
    sub NFACT () { $prod }
}
```

В последнем примере функция NFACT становится подставляемой, потому что у нее пустой прототип, а возвращаемая ей переменная не изменяется этой функцией – более того, ее никто не может изменить, поскольку она находится в лексической области видимости. Поэтому компилятор заменяет вызовы NFACT этим значением, вычисленным на этапе компиляции, потому что она заключена в блок BEGIN.

Если переопределить функцию, которая могла быть сделана подставляемой, обязательно будет выдано предупреждение. (С помощью этого предупреждения можно узнать, что компилятор сделал некоторую подпрограмму подставляемой.) Предупреждение считается достаточно серьезным, чтобы его можно было отключить, т. к. скомпилированные прежде вызовы функции по-прежнему будут использовать ее старое значение. Если требуется переопределить функцию, необходимо обеспечить, чтобы она не сделалась подставляемой, путем удаления прототипа () (что изменяет семантику вызова, поэтому будьте осторожны) или помешав механизму подстановки иным способом, например:

```
sub not_inlined () {
    return 23 if $$;
}
```

Более подробно о том, что происходит в фазах компиляции и выполнения программы, см. в главе 18.

Предосторожности при использовании прототипов

Вероятно, лучше всего применять прототипы с новыми функциями, а не модифицировать старые. Это ведь шаблоны контекста, а не прототипы ANSI C, поэтому особенную осторожность нужно проявлять, чтобы незаметно не изменить контекст. Предположим, например, решено, что функция должна принимать только один параметр, как в следующем примере:

```
sub func ($) {
    my $n = shift;
    print "вы передали мне $n\n";
}
```

Это превращает функцию в унарный оператор (как встроенная rand) и изменяет результат определения аргументов функции компилятором. С новым прототипом функция использует только один аргумент в скалярном контексте вместо многих аргументов в списковом контексте. Если она где-то вызывалась с массивом или списком, даже если этот массив или список содержали единственный элемент, то там, где она раньше нормально работала, теперь мы будем получать совершенно иное:

```
func @foo;                  # считает элементы @foo
func split /:/;             # считает число возвращаемых полей
func "a", "b", "c";         # передает только "a", отбрасывая "b" и "c"
func("a", "b", "c");        # неожиданная ошибка компилятора!
```

Скаляр был неявно передан в начале списка аргументов, что может оказаться несколько неожиданным. Прежний массив @foo, содержавший один элемент, не передается. Вместо него в func передается 1 (число элементов в @foo). А split, будучи вызвана в скалярном контексте, запортит весь ваш список параметров @_. В третьем примере, т. к. func имеет прототип унарного оператора, передано будет только a; затем значение, возвращаемое func, отбрасывается, т. к. оператор «запятая» продолжает вычисление следующих двух элементов и возвращает c. В последнем примере пользователь столкнется во время компиляции с синтаксической ошибкой в коде, который ранее отлично компилировался и выполнялся.

Если в создаваемой новой программе необходимо иметь унарный оператор, который принимает только скалярную переменную, а не прежнее скалярное выражение, можно задать в прототипе, чтобы он принимал *ссылки* на скаляр:

```
sub func (\$) {
    my $nref = shift;
    print "вы передали мне $$nref\n";
}
```

Теперь компилятор не пропустит ничего, что не начинается с символа $:

```
func @foo;              # ошибка компиляции, есть @, нужен $
func split/:/;          # ошибка компиляции, есть функция, нужен $
func $s;                # здесь ok -- получен символ $
```

```
func $a[3];             # и здесь
func $h{stuff}[-1];     # и даже здесь
func 2+5;               # скалярное выражение, ошибка компиляции
func ${ \(2+5) };       # ok, но лекарство оказывается хуже болезни
```

Если не быть внимательным, прототипы могут стать источником неприятностей. Но если вы внимательны, то с их помощью сможете делать отличные вещи. Конечно, это сильное средство, которым, чтобы улучшить наш мир, надо пользоваться в разумных дозах.

Атрибуты подпрограмм

С объявлением или определением подпрограммы может быть связан список атрибутов. Если такой список атрибутов есть, то разделителями в нем служат пробельные символы или двоеточия, и обрабатывается он так, как если бы была встречена директива `use attributes`. Подробности о применении директивы `use attributes` можно найти в главе 31. Есть три стандартных атрибута подпрограмм: `locked`, `method` и `lvalue`.

Атрибуты locked и method

```
# В эту функцию допускается только один поток.
sub afunc : locked { ... }

# В эту функцию для данного объекта допускается только один поток.
sub afunc : locked method { ... }
```

Установка атрибута `locked` имеет смысл только тогда, когда подпрограмма или метод предназначены для вызова одновременно из нескольких потоков. При установке для подпрограммы, не являющейся методом, Perl обеспечивает блокировку самой подпрограммы перед входом в нее. При установке для подпрограммы, которая является методом (т. е. имеющей также атрибут `method`), Perl обеспечивает при каждом ее вызове неявную блокировку первого аргумента (объекта), прежде чем начать ее выполнение.

Семантика этой блокировки такая же, как в случае, если первой командой программы будет применение оператора `lock` к этой подпрограмме. Дополнительно о блокировках читайте в главе 17 «Потоки».

Атрибут `method` может использоваться самостоятельно:

```
sub afunc : method { ... }
```

В настоящее время его действие заключается только в том, что данная подпрограмма не будет вызывать предупреждающее сообщение `Ambiguous call resolved as CORE::%s` (неоднозначный вызов разрешен как `CORE::%s`). (Возможно, когда-нибудь мы дадим ему дополнительный смысл.)

Система атрибутов допускает расширение со стороны пользователя, позволяя ему создавать собственные имена атрибутов. Эти новые атрибуты долж-

ны удовлетворять правилам образования имен простых идентификаторов (никаких знаков пунктуации, кроме символа «_»). К ним может прилагаться список параметров, для которого в данное время выполняется только проверка правильности вложенности скобок.

Вот примеры допустимого синтаксиса (даже если атрибуты неизвестны):

```
sub fnord (&\%) : switch(10,foo(7,3))  :  expensive;
sub plugh () : Ugly('\(") :Bad;
sub xyzzy : _5x5 { ... }
```

Вот примеры неправильного синтаксиса:

```
sub fnord : switch(10,foo();  # нет баланса скобок
sub snoid : Ugly('(');        # нет баланса скобок
sub xyzzy : 5x5;              # "5x5" не является идентификатором
sub plugh : Y2::north;        # "Y2::north" не является простым идентификатором
sub snurt : foo + bar;        # "+" не является двоеточием или пробелом
```

Список атрибутов передается как список строк-констант в код, который связывает их с подпрограммой. Точное представление о том, как он работает (или не работает), можно получить экспериментальным путем. Текущее состояние дел со списками атрибутов и их обработкой можно посмотреть в *attributes*(3).

Атрибут lvalue

Подпрограмма может возвращать модифицируемое скалярное значение, но только если объявить, что она возвращает lvalue:

```
my $val;
sub canmod : lvalue {
    $val;
}
sub nomod {
    $val;
}

canmod() = 5;   # присвоение $val.
nomod()  = 5;   # ERROR
```

При передаче параметров подпрограмме, возвращающей l-значение, обычно нужны скобки для устранения неопределенности типа присваиваемого значения:

```
canmod $x  = 5;     # сначала присваивает $x значение 5!
canmod 42  = 5;     # нельзя изменить константу - ошибка при компиляции
canmod($x) = 5;     # это ok
canmod(42) = 5;     # и это тоже
```

С этим можно справиться в том случае, если подпрограмма принимает один аргумент. Если функция объявлена с прототипом `($)`, то при синтаксичес-

ком анализе она получает приоритет именованного унарного оператора. Поскольку именованные унарные операторы имеют более высокий приоритет, чем присвоение, скобки больше не нужны. (Целесообразность этого зависит от используемого стиля.)

В том случае, если подпрограмма принимает ноль аргументов (т. е. с прототипом `()`), можно, не вызывая неоднозначности, сказать:

```
canmod = 5;
```

Это допустимо, потому что терм не может начинаться с символа `=`. Аналогично можно опускать скобки в вызовах методов, возвращающих l-значение, если им не передаются аргументы:

```
$obj->canmod = 5;
```

Обещаем, что эти две конструкции сохранятся в будущих версиях Perl. Они удобны, когда нужно обернуть (wrap) атрибуты объекта в вызовы методов (чтобы их можно было наследовать, как методы, но обращаться, как к переменным).

Скалярный или списковый контекст подпрограммы, возвращающей l-значение, и правой части присвоения этой подпрограмме определяется так, как если бы вызов подпрограммы был заменен скаляром. Например:

```
data(2,3) = get_data(3,4);
```

Обе подпрограммы здесь вызываются в скалярном контексте, в то время как в:

```
(data(2,3)) = get_data(3,4);
```

и в:

```
(data(2),data(3)) = get_data(3,4);
```

все подпрограммы вызываются в списковом контексте.

В текущей реализации нельзя непосредственно возвращать из подпрограмм с атрибутом lvalue массивы и хеши, но вместо этого всегда можно возвратить ссылку.

7

Форматы

В Perl имеется механизм, позволяющий создавать простые отчеты и диаграммы. Для упрощения этой задачи Perl помогает запрограммировать вывод страницы близко к тому, как она будет выглядеть при печати. Он позволяет следить за такими вещами, как количество строчек на странице, номер текущей страницы, управление выводом заголовков страницы и т. д. Ключевые слова заимствованы из FORTRAN: `format` для объявления и `write` для выполнения; см. соответствующие разделы главы 29 «Функции». К счастью, структура читается значительно легче – скорее как в команде `PRINT USING` языка BASIC. Можете считать, что это *nroff*(1) для бедных. (Это может не быть достаточной рекомендацией для тех, кто знаком с *nroff*.)

Форматы, как пакеты и подпрограммы, не выполняются, а объявляются, поэтому они могут находиться в любом месте программы. (Обычно все же лучше держать их все в одном месте.) У них есть собственное пространство имен, отличное от других типов в Perl. Это значит, что если у нас есть функция с именем `Foo`, то это не то же самое, что формат `Foo`. Однако именем по умолчанию для формата, ассоциированного с некоторым дескриптором файла, служит имя этого дескриптора файла. Поэтому формат по умолчанию для `STDOUT` называется `"STDOUT"`, а формат по умолчанию для дескриптора файла `TEMP` называется `"TEMP"`. Они только выглядят одинаково, но не являются одним и тем же.

Форматы выходных записей объявляются так:

```
format NAME =
FORMLIST
.
```

Если имя *NAME* опущено, то определяется формат STDOUT. *FORMLIST* состоит из последовательности строк, каждая из которых может иметь один из трех типов:

- Комментарий, обозначаемый # в первой колонке.
- Строка шаблона (picture line), определяющая формат одной выводимой строки.
- Строка с аргументами, дающими значения, встраиваемые в предшествующую строку картинки.

Строки шаблонов выводятся точно так, как они выглядят, за исключением того, что вместо некоторых полей подставляются их значения.[1] Каждое поле подстановки в строке шаблона начинается с @ (at) или ^ (крышка). Эти строки не подвергаются никакой интерполяции переменных. Поле @ (не путать с маркером массива @) представляет собой поле обычного типа; поле другого типа, типа ^, предназначено для элементарного заполнения многострочных блоков текста. Размер поля задается заполнением его символами <, > или | для установки, соответственно, выравнивания по левому краю, выравнивания по правому краю и выравнивания по центру. Если переменная превосходит заданную для нее ширину, она усекается.

В качестве альтернативной формы выравнивания по правому краю можно использовать символы # (после начального @ или ^) для задания числового поля. Вместо одного из символов # можно вставить . (точку) и выровнять десятичные точки. Если значение, передаваемое какому-либо из этих полей, содержит символ перевода строки, то выводится только текст перед этим символом. И, наконец, специальное поле @* может служить для вывода многострочных и неусеченных значений; в большинстве случаев оно должно быть единственным полем в строке шаблона.

Значения задаются в следующей строке в том же порядке, как поля в шаблоне. Выражения, служащие источником значений, разделяются запятыми. Все выражения вычисляются в списковом контексте перед обработкой строки, поэтому одно списковое выражение может породить несколько элементов списка. Выражения могут располагаться на нескольких строках, если заключены в фигурные скобки. (При этом открывающая фигурная скобка должна быть первой лексемой первой строки.) Это позволяет выравнивать значения под соответствующими полями формата для облегчения чтения.

Если выражение имеет значением число со знаками после десятичной точки и в соответствующем шаблоне указано, что дробная часть выдачи должна выводиться (т. е. в любом шаблоне, кроме множественных символов #, не со-

---

[1] Даже эти поля поддерживают целостность тех колонок, в которые они вставляются. В строке нет ничего, что заставило бы поля расширяться, сжиматься или сдвигаться в том или ином направлении. Колонки, которые мы видим, священны в смысле WYSIWYG, – исходя из предположения, что используется моноширинный шрифт. Даже для управляющих символов предполагается наличие ширины в один символ.

держащих точки `.`), то символ, выбранный в качестве десятичной точки, определяется текущей установкой `LC_NUMERIC`. Это означает, например, что если в среде окружения этапа исполнения заданы немецкие национальные установки, то вместо точки будет использоваться запятая. Дополнительные сведения имеются на странице руководства *perllocale*.

Внутри выражения пробельные символы `\n`, `\t` и `\f` считаются эквивалентными одному пробелу. Поэтому можно считать, что к каждому значению формата применяется фильтр:

```
$value =~ tr/\n\t\f/ /;
```

Оставшийся пробельный символ, `\r`, вызывает перевод строки, если строка шаблона это позволяет.

Поля шаблона, начинающиеся с символа `^`, а не `@`, обрабатываются особым образом. Поле `#` заполняется пробелами, если значение не определено. Для остальных типов полей крышечка `^` включает режим заполнения. Передаваемое значение должно быть не выражением, а именем скалярной переменной, содержащей строку текста. Perl помещает в поле столько текста, сколько можно, а затем отсекает переднюю часть строки, так что при новом обращении к переменной может быть выведена следующая часть текста. (Да, это означает, что при выполнении вызова `write` переменная не сохраняется и сама изменяется. Сохранить исходное значение можно во временной переменной.) Обычно для вывода текстового блока используют ряд полей, выровненных по вертикали. Можно вывести в последнем поле символы «`...`», которые будут появляться в выводимой информации, если текст слишком длинный, чтобы быть выведенным полностью. Изменить символы, по которым (или после которых) производится разрыв текста, можно, указав в переменной `$:` (если используется модуль `English`, то `$FORMAT_LINE_BREAK_CHARACTERS`) список желаемых символов.

В случае применения полей `^` могут создаваться записи переменной длины. Если форматируемый текст короткий, просто повторите строку формата с полем `^` несколько раз. Для коротких данных это приведет к выводу нескольких пустых строк. Для подавления вывода пустых строк поместите в любом месте строки символ `~` (тильда). (Сама тильда при выводе будет переведена в пробел.) Если рядом с первой тильдой поместить вторую, то строка будет повторяться, пока не исчерпается весь текст в полях этой строки. (Это возможно потому, что поля `^` «съедают» выводимые ими строки. Но если используется поле типа `@` вместе с двумя тильдами, то желательно, чтобы передаваемое в него выражение не выводило одно и то же значение до бесконечности! В этом случае вам нужен оператор `shift` или какой-либо другой оператор с побочным эффектом, позволяющим исчерпать список значений.)

Обработка начала страницы по умолчанию осуществляется форматом, имя которого состоит из имени данного дескриптора файла с присоединенными в конце символами `_TOP`. Она выполняется в начале вывода каждой страницы. См. описание `write` в главе 29.

Вот несколько примеров:

```
# отчет по файлу /etc/passwd
format STDOUT_TOP =

                        Файл Passwd
Имя                 Логин    Офис     Uid   Gid Каталог
-----------------------------------------------------------------
.
format STDOUT =
@<<<<<<<<<<<<<<<<<< @|||||||  @<<<<<<@>>>> @>>>> @<<<<<<<<<<<<<<<<<
$name,              $login,  $office,$uid,$gid, $home
.
# отчет  по бланку сообщения об ошибке
format STDOUT_TOP =
                        Сообщение об ошибке
@<<<<<<<<<<<<<<<<<<<<<<     @|||         @>>>>>>>>>>>>>>>>>>>>>>
$system,                    $%,         $date
-----------------------------------------------------------------
.
format STDOUT =
Сообщение: @<<<<<<<<<<<<<<<<<<
```

Переменные форматов

Имя текущего формата хранится в переменной $~ ($FORMAT_NAME), а имя текущего формата начала страницы – в переменной $^ ($FORMAT_TOP_NAME). Номер текущей выводимой страницы хранится в $% ($FORMAT_PAGE_NUMBER), а число строк на странице – в $= ($FORMAT_LINES_PER_PAGE). Необходимость автоматической очистки буфера этого дескриптора определяется переменной $| ($OUTPUT_AUTOFLUSH). Строка, которую нужно выводить перед началом каждой страницы (кроме первой), хранится в $^L ($FORMAT_FORMFEED). Эти переменные устанавливаются отдельно для каждого дескриптора файла, поэтому нужно выбрать дескриптор файла с помощью select, чтобы изменить переменные его формата:

```
select((select(OUTF),
        $~ = "My_Other_Format",
        $^ = "My_Top_Format"
       )[0]);
```

Довольно безобразно, не так ли? Однако это общепринятая идиома, поэтому не удивляйтесь, увидев ее. Можно использовать временную переменную для хранения предыдущего дескриптора файла:

```
$ofh = select(OUTF);
$~ = "My_Other_Format";
$^ = "My_Top_Format";
select($ofh);
```

В целом такой подход гораздо лучше, поскольку он не только улучшает читаемость, но и создает промежуточную команду в коде, на которой происходит остановка при отладке в пошаговом режиме. При использовании модуля English имена переменных даже становятся читаемыми:

```
use English;
$ofh = select(OUTF);
$FORMAT_NAME     = "My_Other_Format";
$FORMAT_TOP_NAME = "My_Top_Format";
select($ofh);
```

Но курьезные вызовы select все же никуда не исчезли. Для того чтобы избежать их, воспользуйтесь модулем FileHandle, поставляемым с Perl. В этом случае обращаться к специальным переменным можно с помощью методов, в именах которых используются буквы нижнего регистра:

```
use FileHandle;
OUTF->format_name("My_Other_Format");
OUTF->format_top_name("My_Top_Format");
```

Так уже гораздо лучше!

Поскольку в строке значений, следующей за строкой шаблона, могут содержаться произвольные выражения (для полей типа @, не полей ^), можно передать более сложную обработку другим функциям, скажем, sprintf или своим собственным. Например, чтобы вставить в число запятые:

```
format Ident =
    @<<<<<<<<<<<<<<<
    commify($n)
.
```

А для того чтобы вставить в поле реальные @, ~ или ^, сделайте так:

```
format Ident =
Я поместил @ сюда.
          "@"
.
```

Для выравнивания по центру всей строки текста надо сделать примерно нижеследующее:

```
format Ident =
@||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
                            "Какой-то текст"
.
```

Индикатор длины поля > обеспечивает выравнивание текста по правому краю поля, но само поле располагается в точности там, где это указано. Нет прямого способа: «переместить это поле к правому краю страницы, какой бы широкой она ни была». Необходимо указать, где оно должно находиться относительно левого края. Те, кому это крайне необходимо, могут создать формат динамически, задав действительное число колонок, а затем выполнить его через eval:

```
$format  = "format STDOUT = \n"
         . '^' . '<' x $cols . "\n"
         . '$entry' . "\n"
         . "\t^" . "<" x ($cols-8) . "~~\n"
         . '$entry' . "\n"
         . ".\n";
print $format if $Debugging;
eval $format;
die $@ if $@;
```

Самой важной строкой здесь, возможно, является print. Результат вывода print может выглядеть следующим образом:

```
format STDOUT =
^<<<<<<<<<<<<<<
```

Вот маленькая программа, которая работает подобно утилите Unix *fmt*(1):

```
format =
^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< ~~
$_
.
$/ = "";
while (<>) {
    s/\s*\n\s*/ /g;
    write;
}
```

Нижние колонтитулы

В то время как имя текущего формата заголовка хранится в переменной `$^` (`$FORMAT_TOP_NAME`), соответствующего механизма для нижнего колонтитула нет. Одна из главных проблем заключается в том, что размер формата не известен до тех пор, пока он не обработан. Эта задача помещена в список TODO.[1]

Можно предложить такую стратегию: если нижний колонтитул имеет фиксированный размер, то выводить его, проверяя число оставшихся в формате строк `$-` (`$FORMAT_LINES_LEFT`) перед каждой операцией `write`, и при необходимости самостоятельно выводить нижний колонтитул.

Вот другая стратегия: откройте канал с помощью `open(MESELF, "|-")` (см. описание `open` в главе 29) и всегда выполняйте `write` в `MESELF` вместо `STDOUT`. Затем пусть ваш дочерний процесс обработает свой `STDIN` и переставит верхние и нижние колонтитулы так, как вам хочется. Не очень удобно, но реализуемо.

Низкоуровневый доступ к форматированию

Для низкоуровневого доступа к форматированию можно использовать встроенный оператор `formline` и обратиться к `$^A` (переменная `$ACCUMULATOR`) непосредственно. (Форматы, в сущности, компилируются в последовательность вызовов `formline`.) Например:

```
$str = formline <<'END', 1,2,3;
@<<<  @|||  @>>>
END

print "Вау, я запомнил `$^A' в аккумулятор!\n";
```

Чтобы создать подпрограмму `swrite` (которая относится к `write`, как `sprintf` к `printf`), сделайте следующее:

---

[1] Что, конечно, не гарантирует, что мы когда-нибудь это сделаем. Форматы являются, отчасти, пройденным этапом в наш век WWW, Unicode, XML, XSLT и того, что может за ними последовать.

```
use Carp;
sub swrite {
    croak "использование: swrite PICTURE ARGS" unless @_;
    my $format = shift;
    $^A = "";
    formline($format, @_);
    return $^A;
}

$string = swrite(<<'END', 1, 2, 3);
Check me out
@<<<  @|||  @>>>
END

print $string;
```

Если используется модуль `FileHandle`, то с помощью `formline` можно переносить блок текста по колонке 72 так:

```
use FileHandle;
STDOUT->formline("^" . ("<" x 72) . "~~\n", $long_text);
```

8

Ссылки

Как по практическим, так и по философским причинам Perl всегда тяготел к плоским линейным структурам данных. А для многих задач это как раз то, что требуется.

Допустим, требуется построить простую таблицу (двумерный массив) с демографической статистикой – возраст, цвет глаз и вес – для группы людей. Можно сделать это, сначала создав массив для каждого отдельного лица:

```
@john = (47, "темно-карие", 186);
@mary = (23, "карие", 128);
@bill = (35, "голубые",  157);
```

Затем можно создать один дополнительный массив, содержащий имена остальных массивов:

```
@vitals = ('john', 'mary', 'bill');
```

Чтобы изменить цвет глаз Джона на «красные» после ночи, проведенной в увеселительных местах, нам надо иметь возможность изменить содержание массива `@john`, используя только простую строку `"john"`. Это основная проблема *косвенной адресации* (*indirection*), которая в различных языках решается по-разному. В C самой распространенной формой косвенной адресации является указатель, который позволяет одной переменной хранить в памяти адрес другой переменной. В Perl самая распространенная форма косвенной адресации – *ссылка* (*reference*).

Что такое ссылка?

В нашем примере `$vitals[0]` имеет значение `"john"`. Это значит, что данный элемент содержит строку, которая представляет собой имя другой (глобальной) переменной. Мы говорим, что первая переменная *ссылается* на вторую, и эта ссылка называется *символической* (*symbolic*), поскольку Perl нужно найти `@john` в таблице символов. (Символические ссылки на переменные аналогичны символическим ссылкам файловой системы.) О символических ссылках мы будем говорить далее в этой главе.

Другим типом ссылки является *жесткая* (*hard*) ссылка, и именно к ним прибегает большинство программистов Perl для осуществления косвенной адресации. Жесткими мы называем эти ссылки потому, что они надежные и прочные. Если хотите, можете считать жесткие ссылки настоящими ссылками, а символические ссылки – поддельными. Разница такая же, как между настоящей дружбой и похвальбой знакомством с известным человеком. Если мы не указываем, какого типа ссылку имеем в виду, значит, это жесткая ссылка. На рис. 8.1 изображена переменная с именем `$bar`, которая ссылается на содержимое скаляра с именем `$foo`, имеющего значение `bot`.

Рис. 8.1. Жесткая ссылка и символическая ссылка

В отличие от символических ссылок, настоящая ссылка ссылается не на имя другой переменной (которое просто является контейнером для значения), а на фактическое значение, некоторый внутренний фрагмент данных. Подходящего слова для него нет, но когда приходится обозначать его, мы называем его *объектом ссылки* (*referent*). Предположим, например, что создается жесткая ссылка на массив с лексической областью видимости, который называется `@array`. Эта жесткая ссылка и объект, на который она ссылается, будут существовать даже тогда, когда `@array` выйдет из области видимости. Объект ссылки будет уничтожен только тогда, когда будут ликвидированы все ссылки на него.

У объекта ссылки в действительности нет собственного имени, помимо ссылки на него. Иными словами, имя каждой переменной Perl «живет» в некоторой таблице имен, содержащей одну жесткую ссылку на относящийся к нему (и в других отношениях безымянный) объект ссылки. Этот объект ссылки может быть простым – например, числом или строкой, либо сложным –

например, массивом или хешем. В любом случае есть ровно одна ссылка из переменной на его значение. Можно создать и дополнительные ссылки на тот же самый объект ссылки, но в таком случае переменная не будет об этом знать (или беспокоиться).[1]

Символическая ссылка – это просто строка, служащая именем для некоторого элемента таблицы имен в пакете. Это не столько какой-то особый тип, сколько способ обращения со строкой. Однако жесткая ссылка – это зверь совсем иной породы. Это третий из трех видов фундаментальных скалярных типов данных. Чтобы сослаться на нечто, жесткой ссылке не нужно знать его имя, и, самое главное, совершенно нормально, если такого имени, которое можно использовать, и нет. Такие совершенно безымянные объекты ссылки называются *анонимными*; мы расскажем о них далее в разделе «Анонимные данные».

В терминологии данной главы, чтобы *ссылаться* (*to reference*) на значение, нужно создать на него жесткую ссылку. (Для этого акта созидания есть специальный оператор.) Создаваемая при этом ссылка является просто скаляром, который во всех известных контекстах ведет себя как всякий другой скаляр. *Разыменовать* (*dereference*) этот скаляр означает использовать ссылку для доступа к объекту ссылки. Как создание ссылки, так и разыменование происходят только при вызове определенного явного механизма; неявные создание ссылки и разыменование в Perl никогда не происходят. Ну, или почти никогда.

В вызове функции *может* использоваться семантика неявной передачи по ссылке, если это объявлено соответствующим образом в ее прототипе. В этом случае при вызове функции ссылка не передается ей явным образом, но, тем не менее, ее приходится явно разыменовывать в теле функции. См. раздел «Прототипы» в главе 6. И, если уж быть совсем честным, то некоторое закулисное разыменование происходит, когда используются некоторые виды дескрипторов файлов, но это делается для поддержки обратной совместимости и происходит прозрачным для обычного пользователя способом. И, наконец, две встроенные функции, `bless` и `lock`, принимают в качестве своих аргументов ссылку и неявным образом разыменовывают ее, чтобы применить свои механизмы к тому, что за ней скрывается. Но если оставить в стороне эти признания, то базовый принцип все же состоит в том, что Perl не заинтересован во внесении беспорядка в ваши уровни косвенной адресации.

Ссылка может указывать на любую структуру данных. Поскольку ссылки являются скалярами, их можно хранить в массивах и хешах, создавая таким образом массивы массивов, массивы хешей, хеши массивов, массивы хешей и функций и т. д. Примеры тому есть в главе 9 «Структуры данных».

Имейте, однако, в виду, что внутреннее представление массивов и хешей в Perl является одномерным. Это означает, что их элементы могут хранить только скалярные значения (строки, числа и ссылки). Говоря «массив мас-

---

[1] Если вам это интересно – определить счетчик ссылок для объекта можно с помощью модуля `Devel::Peek`, поставляемого с Perl.

сивов», мы в действительности имеем в виду «массив ссылок на массивы» – так же, как, говоря «хеш функций», мы в действительности имеем в виду «хеш ссылок на подпрограммы». Но поскольку ссылки являются единственным способом реализации таких структур в Perl, то использование более короткого, но менее точного выражения не настолько неточно, чтобы быть неверным, и потому его не следует совершенно отвергать, если только вы не придаете особого внимания такого рода вещам.

Создание ссылок

Есть несколько способов создания ссылок, большинство из которых мы опишем, а затем объясним, как использовать (разыменовывать) получаемые ссылки.

Оператор обратной косой черты

Ссылку на любую именованную переменную или подпрограмму можно создать с помощью обратной косой черты. (Можно также использовать ее с анонимными скалярными значениями, например, `7` или `"camel"`, но вам не часто это понадобится.) Этот оператор работает аналогично оператору `&` (адрес) в C, по крайней мере, на первый взгляд.

Вот несколько примеров:

```
$scalarref = \$foo;
$constref  = \186_282.42;
$arrayref  = \@ARGV;
$hashref   = \%ENV;
$coderef   = \&handler;
$globref   = \*STDOUT;
```

Оператор обратной косой черты может сделать больше, чем просто создать одну ссылку. Если применить его к списку, то он создаст целый список ссылок. Подробности см. в разделе «Другие приемы использования жестких ссылок».

Анонимные данные

В только что приведенных примерах оператор обратной косой черты просто создает дубликат ссылки, уже содержащейся в имени переменной, за одним исключением. На `186_282.42` именованная переменная не ссылается, это просто значение. Это пример одного из *анонимных* объектов ссылки, уже упоминавшихся выше. Доступ к анонимным объектам ссылки осуществляется только с помощью ссылок. Данный объект представляет собой число, но можно также создавать анонимные массивы, хеши и подпрограммы.

Формирователь анонимных массивов

Ссылку на анонимный массив можно создать с помощью квадратных скобок:

```
$arrayref = [1, 2, ['a', 'b', 'c', 'd']];
```

Здесь мы сформировали анонимный массив из трех элементов, последним элементом которого является ссылка на анонимный массив из четырех элементов (изображенный на рис. 8.2). (Для доступа к этим элементам может использоваться многомерный синтаксис, описываемый ниже. Например, `$arrayref->[2][1]` имеет значение `'b'`.)

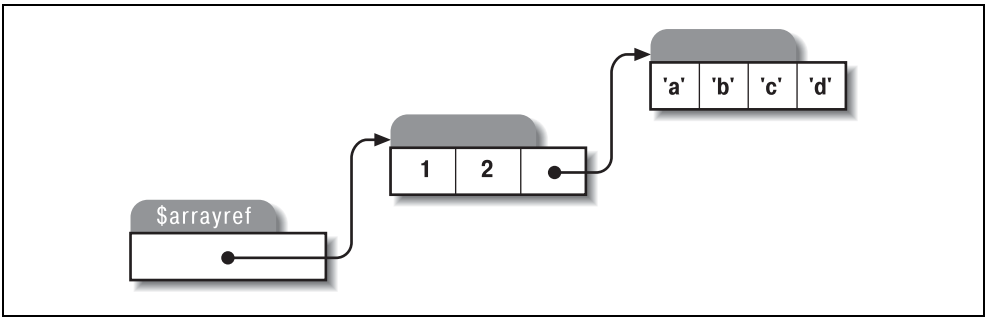


Рис. 8.2. Ссылка на массив, третий элемент которого сам является ссылкой на массив

Теперь у нас есть способ представить таблицу, приведенную в начале главы:

```
$table = [ [ "john", 47, "темно-карие", 186],
           [ "mary", 23, "карие", 128],
           [ "bill", 35, "голубые",  157] ];
```

Квадратные скобки действуют подобным образом только там, где анализатор Perl предполагает наличие терма в выражении. Не следует путать их с квадратными скобками в выражении типа `$array[6]`, хотя мнемоническая ассоциация с массивами является намеренной. Внутри строки, заключенной в двойные кавычки, квадратные скобки не образуют анонимных массивов, а становятся литеральными символами строки. (Квадратные скобки все же действуют при создании индексов в строках, иначе нельзя было бы выводить строковые значения вида `"VAL=$array[6]\n"`. И если уж совсем откровенно, то, на самом деле, можно протащить формирователи анонимных массивов в строки, но только если они встроены в более крупное выражение, которое подвергается интерполяции. Об этой возможности речь пойдет ниже, поскольку она включает в себя не только создание ссылки, но и разыменование.)

Формирователь анонимных хешей

Ссылку на анонимный хеш можно создать с помощью фигурных скобок:

```
$hashref = {
    'Адам'   => 'Ева',
```

```
    'Клайд'  => $bonnie,
    'Антоний' => 'Клео' . 'патра',
};
```

В значениях (но не ключах) хеша можно смело смешивать другие формирователи анонимных массивов, хешей и подпрограмм, создавая сколь угодно сложные структуры.

Теперь у нас есть еще один способ представить таблицу, приведенную в начале главы:

```
$table = {
            "john" => [ 47, "темно-карие", 186 ],
            "mary" => [ 23, "карие", 128 ],
            "bill" => [ 35, "голубые",  157 ],
};
```

Это хеш массивов. Выбор наилучшей структуры данных является непростым делом, и этому вопросу будет посвящена глава 9. В качестве рекламы продемонстрируем использование хеша хешей для представления нашей таблицы:

```
$table = {
           "john" => { age    => 47,
                       eyes   => "темно-карие",
                       weight => 186,
                     },
           "mary" => { age    => 23,
                       eyes   => "карие",
                       weight => 128,
                     },
           "bill" => { age    => 35,
                       eyes   => "голубые",
                       weight => 157,
                     },
 };
```

Как и квадратные, фигурные скобки действуют подобным образом только там, где анализатор Perl предполагает наличие терма в выражении. Не следует путать их с фигурными скобками в выражении типа `$hash{key}` — хотя мнемоническая ассоциация с массивами и в этом случае является намеренной. Те же предостережения относятся к использованию фигурных скобок в строках.

Есть и дополнительное предостережение, которое не было актуальным для квадратных скобок. Поскольку фигурные скобки предназначены для ряда других целей (в том числе блоков), иногда приходится устранять неоднозначность фигурных скобок в начале команды, помещая перед ними + или `return`, чтобы Perl понял, что открывающая скобка не начинает блок. Например, чтобы функция создавала новый хеш и возвращала ссылку на него, можно использовать такие варианты:

```
sub hashem {        { @_ } }   # НЕВЕРНО, но без сообщений - - возвращает @_.
sub hashem {       +{ @_ } }   # Ok.
sub hashem { return { @_ } }   # Ok.
```

Формирователь анонимных подпрограмм

Создать ссылку на анонимную подпрограмму можно с помощью объявления `sub` без имени подпрограммы:

```
$coderef = sub { print "Boink!\n" };  # Теперь &$coderef выводит "Boink!"
```

Обратите внимание на точку с запятой, которая требуется для завершения выражения. (Точка с запятой не требуется при более традиционном использовании `sub` *`NAME`* `{}` для объявления и определения именованной подпрограммы.) Безымянная `sub {}` представляет собой не столько объявление, сколько оператор – как `{}` или `eval {}` – за исключением того, что находящийся внутри код не выполняется немедленно. Вместо этого генерируется ссылка на код, в нашем примере помещаемая в `$coderef`. Однако сколько бы раз ни выполнялась приведенная выше строка, `$coderef` будет ссылаться на одну и ту же безымянную процедуру.[1]

Конструкторы объектов

Подпрограммы также могут возвращать ссылки. Может звучать банально, но иногда мы *обязаны* использовать для создания ссылки подпрограмму, а не создавать ссылку самостоятельно. В частности, особые подпрограммы, называемые *конструкторами*, создают или возвращают ссылки на объекты. Объект – это просто ссылка особого типа, которой известно, с каким классом она связана, и конструкторы умеют создавать эту связь. Они делают это, беря обычный предмет ссылки и превращая его в объект с помощью оператора `bless`, поэтому можно считать, что объект – это «blessed reference» («освященная», или обработанная, ссылка). Ничего религиозного здесь не происходит. Поскольку класс выступает как тип, определенный пользователем, обработка объекта ссылки с помощью `bless` просто делает его типом, определенным пользователем, в дополнение к встроенному. Конструкторам часто дают имя `new` – особенно в программах на C++, – но в Perl их можно называть как угодно.

Конструкторы можно вызывать любым из следующих способов:

```
$objref = Doggie::->new(Tail => 'короткий', Ears => 'длинные');  #1
$objref = new Doggie:: Tail => 'короткий', Ears => 'длинные';    #2
$objref = Doggie->new(Tail => 'короткий', Ears => 'длинные');    #3
$objref = new Doggie Tail => 'короткий', Ears => 'длинные';      #4
```

[1] Несмотря на то что анонимная подпрограмма только одна, она может использовать несколько экземпляров лексических переменных в зависимости от того, когда генерируется ссылка на подпрограмму. О них рассказывается далее в разделе «Замыкания».

Первый и второй вызовы одинаковы. В обоих вызывается функция с именем new, которую предоставляет модуль Doggie. Третий и четвертый вызовы такие же, как первые два, но несколько более двусмысленны: анализатор запутается, если вы определите собственную программу с именем Doggie. (Почему обычно и придерживаются нижнего регистра в именах подпрограмм и верхнего регистра в именах модулей.) Четвертый вызов может также сбиться, если определить собственную подпрограмму new и не выполнить команды require или use для модуля Doggie, каждая из которых имеет эффект объявления модуля. Всегда объявляйте свои модули, если хотите пользоваться вариантом 4. (И остерегайтесь «бродячих» подпрограмм Doggie.)

Обсуждение объектов Perl см. в главе 12 «Объекты».

Ссылки на дескрипторы

Ссылки на дескрипторы файлов или дескрипторы каталогов можно создавать посредством ссылки на typeglob с тем же именем:

```
splutter(\*STDOUT);

sub splutter {
    my $fh = shift;
    print $fh "her um well a hmmm\n";
}

$rec = get_rec(\*STDIN);
sub get_rec {
    my $fh = shift;
    return scalar <$fh>;
}
```

Если передаются дескрипторы файлов, то можно применять для этого голые (простые) typeglob: в вышеприведенном примере вместо *STDOUT и *STDIN можно было использовать *STDOUT или *STDIN.

Хотя typeglob и ссылки на typeglob обычно можно использовать взаимозаменяемым образом, в некоторых случаях это недопустимо. Простые typeglob нельзя превратить в объекты с помощью bless, а ссылки на typeglob нельзя возвращать из области видимости локализованного typeglob.

При создании новых дескрипторов файлов раньше использовали примерно такой код для открытия списка файлов:

```
for $file (@names) {
    local *FH;
    open(*FH, $file) || next;
    $handle{$file} = *FH;
}
```

Такой код по-прежнему допустим, но сейчас можно легко позволить неопределенной переменной автоматически оживить анонимный typeglob:

```
for $file (@names) {
    my $fh;
    open($fh, $file) || next;
    $handle{$file} = $fh;
}
```

Для косвенных дескрипторов файлов не важно, применяется ли typeglob, ссылки на typeglob или какой-нибудь более экзотический объект ввода/вывода. Вы просто используете скаляр, который так или иначе будет интерпретирован как дескриптор файла. В большинстве случаев почти никакого значения не имеет, что выбрать – typeglob или ссылку на typeglob. Как уже сообщалось раньше, при этом неявно работают некоторые механизмы разыменования.

Ссылки на таблицы символов

В некоторых редких случаях во время написания программы может быть неизвестно, какого типа ссылка требуется. Ссылку можно создать с помощью особого синтаксиса, который любовно называют синтаксисом `*foo{THING}`. Синтаксис `*foo{THING}` возвращает ссылку на позицию для *THING* в `*foo`,[1] которая является записью в таблице имен, содержащей значения `$foo`, `@foo`, `%foo` и компании.

```
$scalarref = *foo{SCALAR};   # То же, что и \$foo
$arrayref  = *ARGV{ARRAY};   # То же, что и \@ARGV
$hashref   = *ENV{HASH};     # То же, что и \%ENV
$coderef   = *handler{CODE}; # То же, что и \&handler
$globref   = *foo{GLOB};     # То же, что и \*foo
$ioref     = *STDIN{IO};     # Э-э...
```

Все они говорят сами за себя, за исключением `*STDIN{IO}`. Последний возвращает фактический внутренний объект `IO::Handle`, содержащийся в typeglob, то есть ту часть typeglob, в которой реально заинтересованы различные функции ввода/вывода. Для совместимости с более ранними версиями Perl обозначение `*foo{FILEHANDLE}` служит синонимом для более знакомого `*foo{IO}`.

Теоретически, использовать `*HANDLE{IO}` можно там же, где и `*HANDLE` или `\*HANDLE`, например, при передаче дескрипторов в подпрограмму или из нее или для запоминания их в более крупных структурах данных. (На практике все же есть некоторые шероховатости, которые нужно сгладить.) Преимущество заключается в возможности применения с реально требуемыми объектами ввода/вывода, а не со всем typeglob, что позволяет избежать риска затереть полезные данные, присваивая значение переменной typeglob (хотя если всегда выполнять присваивание скалярной переменной, а не typeglob,

[1] Развернутое толкование и историю термина «foo» можно найти в Файле Жаргона (*http://www.jargon.org/*). – *Примеч. ред.*

то все будет в порядке). Недостаток состоит в отсутствии в настоящее время возможности самооживления.[1]

```
splutter(*STDOUT);
splutter(*STDOUT{IO});

sub splutter {
    my $fh = shift;
    print $fh "her um well a hmmm\n";
}
```

Оба вызова `splutter()` выводят `"her um well a hmmm"`.

Конструкция `*foo{THING}` возвращает `undef`, если указанный аргумент *THING* компилятор еще не видел, за исключением тех случаев, когда *THING* представляет собой `SCALAR`. Дело в том, что `*foo{SCALAR}` возвращает ссылку на анонимный скаляр, даже если компилятор еще не видел `$foo`. (Perl всегда добавляет скаляр к любому typeglob для оптимизации кода по размеру. Но не следует полагать, что так будет и в последующих версиях.)

Неявное создание ссылок

Заключительный метод создания ссылок, в сущности, методом и не является. Ссылки надлежащего типа возникают при разыменовании их в контексте l-значения, предполагающем их существование. Это очень удобно и представляет собой как раз то, чего вы ожидаете. Эта тема будет обсуждаться далее в этой главе, когда мы расскажем, как разыменовывать все эти созданные нами ссылки.

Использование жестких ссылок

Так же как есть много способов создания ссылок, есть много способов их использования, или *разыменования* (*dereference*). Есть только один главный принцип: Perl не выполняет неявно создание ссылок или их разыменование.[2] Если в скаляре содержится ссылка, то он ведет себя как простой скаляр. Он не начинает при этом чудесным образом вести себя как массив, или хеш, или подпрограмма: об этом ему нужно сказать явным образом путем разыменования.

Использование переменной в качестве имени переменной

Встретив скаляр, скажем, `$foo`, следует представить себе «скалярное значение `foo`». Это значит, что в таблице имен есть запись `foo`, а разыменовываю-

---

[1] В настоящее время `open my $fh` самооживляет typeglob, а не объект `IO::Handle`, но когда-нибудь мы, возможно, исправим это, поэтому не следует полагаться на то, что самооживляемое `open` в настоящее время будет typeglob. См. глоссарий.

[2] Мы уже признавались в том, что это маленькая ложь. Делать это снова мы не собираемся.

щий символ $ дает возможность посмотреть на скалярное значение, которое в ней находится. Если то, что в ней находится, представляет собой ссылку, то можно посмотреть, что находится *в этой* ссылке, поставив впереди еще один разыменовывающий символ и разыменовав таким образом $foo. Или, если посмотреть на это с другой стороны, можно заменить скалярной переменной, указывающей на фактический объект ссылки, литерал foo в скаляре $foo. Это справедливо для переменной любого типа, поэтому не только $$foo – это скалярное значение того, на что ссылается $foo, но и @$bar – это значение типа массив – того, на что ссылается $bar, и %$glarch – это значение типа хеш – того, на что ссылается $glarch, и т. д. Вывод таков: можно поместить дополнительный разыменовывающий символ впереди любой простой скалярной переменной и в результате разыменовать ее:

```
$foo         = "трехгорбый";
$scalarref   = \$foo;         # $scalarref теперь ссылка на $foo
$camel_model = $$scalarref;   # $camel_model теперь"трехгорбый"
```

Вот еще несколько примеров разыменования:

```
$bar = $$scalarref;

push(@$arrayref, $filename);
$$arrayref[0] = "January";            # Установить первый элемент @$arrayref
@$arrayref[4..6] = qw/May June July/; # Установить несколько элементов @$arrayref

%$hashref = (KEY => "ЗВЕНИТ", BIRD => "ПОЕТ");  # Инициализировать весь хеш
$$hashref{KEY} = "ЗНАЧЕНИЕ";                   # Установить одну пару ключ/значение
@$hashref{"КЛЮЧ1","КЛЮЧ2"} = ("ЗНАЧ1","ЗНАЧ2");   # Установить еще две пары
&$coderef(1,2,3);

print $handleref "вывод\n";
```

В этом виде разыменования могут применяться только к простым скалярным переменным (без индексов). Это значит, что разыменование происходит *до* (или связывает сильнее) поиска в массиве или хеше. Попробуем использовать фигурные скобки, чтобы пояснить, что мы имеем в виду: выражение вида $$arrayref[0] эквивалентно ${$arrayref}[0] и означает первый элемент массива, на который ссылается $arrayref. Это совсем не то же самое, что ${$arrayref[0]}, разыменовывающее первый элемент массива с именем @arrayref, возможно, несуществующего. Аналогично, $$hashref{KEY} представляет то же самое, что и ${$hashref}{KEY}, и не имеет отношения к ${$hashref{KEY}}, разыменовывающему элементу хеша с именем %hashref, возможно, несуществующего. Пока вы этого не поймете, вам будет плохо.

Конкатенируя соответствующие разыменовывающие символы, можно достигать разных уровней ссылок и разыменований. Следующий пример выводит "howdy":

```
$refrefref = \\\"howdy";
print $$$$refrefref;
```

Можно считать, что знаки доллара действуют слева направо. Однако в начале цепочки все же должна быть простая скалярная переменная без индекса. Есть, правда, более замысловатый способ, которым мы уже тайком пользовались раньше и о котором расскажем в следующем разделе.

Использование блока в качестве имени переменной

Можно разыменовывать не только имя простой переменной, но и содержимое блока. Всюду, где буквенно-цифровой идентификатор стоит как часть имени переменной или подпрограммы, можно заменить идентификатор блоком, возвращающим ссылку правильного типа. Иными словами, все приведенные выше примеры можно избавить от двусмысленности следующим образом:

```
$bar = ${$scalarref};
push(@{$arrayref}, $filename);
${$arrayref}[0] = "January";
@{$arrayref}[4..6] = qw/May June July/;
${$hashref}{"KEY"} = "ЗНАЧЕНИЕ";
@{$hashref}{"КЛЮЧ1","КЛЮЧ2"} = ("ЗНАЧ1","ЗНАЧ2");
&{$coderef}(1,2,3);
```

не говоря уже о:

```
$refrefref = \\\"howdy";
print ${${${$refrefref}}};
```

Положим, в таких простых примерах глупо использовать фигурные скобки, но блок может содержать произвольное выражение – в частности, выражения с индексами. В следующем примере предполагается, что `$dispatch{$index}` содержит ссылку на подпрограмму (такие ссылки иногда называют «coderef» – ссылкой на код). Подпрограмма вызывается здесь с тремя аргументами.

```
&{ $dispatch{$index} }(1, 2, 3);
```

Здесь использование блока необходимо. Без внешней пары фигурных скобок Perl рассматривал бы это как ссылку на код `$dispatch`, а не `$dispatch{$index}`.

Использование оператора «стрелка»

Для ссылок на массивы, хеши или подпрограммы предназначен третий способ разыменования с использованием инфиксного оператора `->`. Эта форма синтаксической поблажки облегчает доступ к отдельным элементам массивов и хешей, а также косвенный вызов подпрограмм.

Тип разыменования определяется правым операндом, т. е. тем, что следует непосредственно за стрелкой. Если за стрелкой следует квадратная или фигурная скобка, то левый операнд рассматривается как ссылка на массив или хеш, соответственно, индексом для которых является выражение в правой

части. Если за стрелкой следует левая круглая скобка, то левый операнд рассматривается как ссылка на подпрограмму, которая должна вызываться с параметрами, передаваемыми в круглых скобках справа.

Команды в каждой из следующих триад эквивалентны между собой, соответствуя трем типам обозначений, с которыми мы познакомились. (Эквивалентные элементы выровнены при помощи пробелов.)

```
$  $arrayref  [2] = "Dorian";          #1
${ $arrayref }[2] = "Dorian";          #2
   $arrayref->[2] = "Dorian";          #3

$  $hashref  {KEY} = "F#major";        #1
${ $hashref }{KEY} = "F#major";        #2
   $hashref->{KEY} = "F#major";        #3

&  $coderef  (Presto => 192);          #1
&{ $coderef }(Presto => 192);          #2
   $coderef->(Presto => 192);          #3
```

Можно заметить, что начальный разыменовывающий символ отсутствует в третьем обозначении каждой тройки. Perl угадывает начальный символ, который нельзя поэтому использовать для разыменования целых массивов, целых хешей или их срезов. Однако, работая со скалярными выражениями, можно расположить слева от `->` любое выражение, в том числе другое разыменование, поскольку операторы стрелки связываются слева направо:

```
print $array[3]->{"English"}->[0];
```

Из этого выражения можно заключить, что четвертый элемент `@array` должен быть ссылкой на хеш, а значением элемента `"English"` в этом хеше должна быть ссылка на массив.

Обратите внимание на то, что `$array[3]` и `$array->[3]` — это не одно и то же. В первом случае речь идет о четвертом элементе массива `@array`, а во втором — о четвертом элементе массива (возможно, анонимного), ссылка на который содержится в `$array`.

Предположим теперь, что элемент `$array[3]` не определен. Следующая команда все же допустима:

```
$array[3]->{"English"}->[0] = "January";
```

Это один из тех упоминавшихся ранее случаев, когда ссылка создается (самооживляется) при использовании в качестве l-значения (т. е. когда ей присваивается значение). Если элемент `$array[3]` был неопределенным, он автоматически определяется как ссылка на хеш, и поэтому в него можно поместить значение `$array[3]->{"English"}`. После этого `$array[3]->{"English"}` автоматически определяется как ссылка на массив, и можно что-нибудь присвоить первому элементу этого массива. Обратите внимание, что с r-значениями дело обстоит несколько иначе: `print  $array[3]->{"English"}->[0]`

определяет только $array[3] и $array[3]->{"English"}, но не $array[3]->{"English"}->[0], поскольку последний элемент не является l-значением. (То, что он вообще определяет первые два элемента в контексте r-значения, может считаться программной ошибкой. Возможно, мы это исправим.)

Использование стрелки между квадратными или круглыми скобками либо между закрывающей квадратной или фигурной скобкой и круглой скобкой при косвенном вызове подпрограммы не является обязательным. Предшествующий код можно укоротить до следующего:

```
$dispatch{$index}(1, 2, 3);
$array[3]{"English"}[0] = "January";
```

Для обычных массивов это дает многомерные массивы, такие же, как массивы в C:

```
$answer[$x][$y][$z] += 42;
```

Ну, хорошо, *не совсем* такие, как в C. Во-первых, C не умеет увеличивать свои массивы при необходимости, а Perl умеет. Кроме того, некоторые конструкции, выглядящие аналогично, анализируются в обоих языках по-разному. В Perl следующие две команды делают одно и то же:

```
$listref->[2][2] = "hello";    # Вполне понятно
$$listref[2][2]  = "hello";    # Несколько смущает
```

Вторая команда может привести в замешательство программиста на C, привыкшего использовать *a[i] для задания того, на что указывает *i*-й элемент a. Но в Perl пять символов ($ @ * % &) осуществляют более сильное связывание, чем фигурные или квадратные скобки.[1] Поэтому $$listref, а не $listref[2] принимается за ссылку на массив. Для того чтобы было как в C, нужно либо написать ${$listref[2]} и заставить $listref[2] выполняться перед разыменованием $ либо использовать обозначение ->:

```
$listref[2]->[$greeting] = "hello";
```

Использование методов объектов

Если ссылка оказывается ссылкой на объект, то класс, определяющий этот объект, может предоставлять методы для доступа к внутреннему содержанию объекта, и обычно нужно придерживаться этих методов, если вы только используете этот класс (а не реализуете его). Иными словами, ведите себя хорошо и не обращайтесь с объектом, как с обычной ссылкой, несмотря на то, что Perl позволяет делать это в случае действительной необходимости.

---

[1] Это не связано с приоритетами операторов. Разыменовывающие символы в Perl не являются операторами в этом смысле. Грамматика Perl просто запрещает чему-либо более сложному, чем простая переменная или блок, следовать начальному разыменовывающему символу – по разным причинам.

Perl не обеспечивает инкапсуляцию. Мы не сторонники тоталитаризма. Однако мы рассчитываем на некоторую элементарную вежливость.

В обмен на эту вежливость вы получаете полную ортогональность между объектами и структурами данных. Любая структура данных может вести себя как объект, когда это требуется. Или не вести, когда не требуется.

Псевдохеши

Псевдохеш – это любая ссылка на массив, первым элементом которого является ссылка на хеш. С псевдохешем можно обращаться либо как со ссылкой на массив (что естественно), либо как со ссылкой на хеш (что может быть неожиданным). Вот пример псевдохеша:

```
$john = [ {age => 1, eyes => 2, weight => 3}, 47, "темно-карие", 186 ];
```

Хеш, на который ссылается `$john->[0]`, определяет имена (`"age"`, `"eyes"`, `"weight"`) последующих элементов массива (47, `"темно-карие"`, 186). Теперь обратиться к элементу можно с помощью любой из нотаций – хеша и массива:

```
$john->{weight}             # Обращается к $john как к ссылке на хеш
$john->[3]                  # Обращается к $john как к ссылке на массив
```

Механизм работы псевдохеша лежит неглубоко; он знает только один «фокус»: как преобразовать разыменование хеша в разыменование массива. При добавлении в псевдохеш нового элемента необходимо явно сообщить хешу, осуществляющему соответствие, место, в котором будет находиться элемент, прежде чем можно будет воспользоваться хеш-нотацией:

```
$john->[0]{height} = 4;     # height будет элементом 4
$john->{height} = "tall";   # Или $john->[4] = "tall"
```

Perl возбудит исключительную ситуацию, если попытаться удалить ключ из псевдохеша, а вот из хеша соответствия ключи можно удалить всегда. Perl возбуждает исключительную ситуацию и при попытке доступа к несуществующему ключу, где «существование» означает присутствие в хеше соответствия:

```
delete $john->[0]{height};  # Удаляет только из хеша
$john->{height};            # Теперь вызовет исключительную ситуацию
$john->[4];                 # По-прежнему выводит "tall"
```

Не пытайтесь сращивать массив, если не очень хорошо представляете, что вы делаете. Когда элементы массива перемещаются, значения в хеше соответствия продолжают ссылаться на *прежние* позиции элементов, если только не изменить явным образом и их. Механизм псевдохеширования не всемогущ.

Чтобы избежать несоответствий, можно использовать для создания псевдохешей функцию `fields::phash`, предоставляемую директивой `use fields`:

```
use fields;
$ph = fields::phash(age => 47, eyes => "коричневые", weight => 186);
print $ph->{age};
```

Есть два способа проверки существования ключа в псевдохеше. В первом способе используется функция exists, с помощью которой проверяется, устанавливалось ли данное поле. Такое действие соответствует поведению реального хеша. Например:

```
use fields;
$ph= fields::phash([qw(age eyes brown)], [47]);
$ph->{eyes} = undef;

print exists $ph->{age};     # True, 'age' установлен по объявлению.
print exists $ph->{weight};  # False, 'weight' не использовался.
print exists $ph->{eyes};    # True, было присвоение 'eyes'.
```

Второй способ – использовать exists с хешем соответствия, находящимся в первом элементе массива. При этом проверяется, является ли данный ключ допустимым полем для этого псевдохеша:

```
print exists $ph->[0]{age};   # True, 'age' является допустимым полем
print exists $ph->[0]{name};  # False, 'name' нельзя использовать
```

В отличие от того, что происходит в настоящем хеше, вызов delete для элемента псевдохеша удаляет только значение массива, соответствующее ключу, а не действительный ключ в хеше соответствия. Чтобы удалить ключ, нужно явно удалить его из хеша соответствия. После этого использовать имя данного ключа в индексе псевдохеша уже нельзя:

```
print delete $ph->{age};     # Удаляет и возвращает $ph->[1], 47
print exists $ph->{age};     # Теперь false
print exists $ph->[0]{age};  # True, ключ 'age' еще есть
print delete $ph->[0]{age};  # Теперь ключа 'age' нет
print $ph->{age};            # Ошибка на этапе исполнения
```

Возможно, читатель недоумевает, для чего нужен этот маскарад массивов, рядящихся в одежды хешей. Массивы предоставляют более быстрый поиск и более эффективное хранение, тогда как хеши предлагают удобство доступа к данным по имени, а не по номеру; псевдохеши соединяют в себе достоинства того и другого. Но самые большие преимущества становятся очевидны на стадии компиляции Perl. С помощью одной-двух директив компилятор может проверить правильность обращения к допустимым полям, и вы обнаружите несуществующие индексы (или ошибки написания) еще до того, как программа будет запущена.

Такие свойства псевдохешей, как скорость, эффективность и проверка доступности на этапе компиляции (можно рассматривать ее как контроль типов), особенно удобны при создании эффективных и надежных модулей классов. См. обсуждение директивы use fields в главе 12 «Объекты» и в главе 31 «Модули прагм».

Псевдохеши являются новой и относительно экспериментальной функцией, поэтому их реализация вполне может подвергнуться изменениям в будущем. Чтобы не пострадать от таких изменений, всегда используйте документированный интерфейс модуля `fields` с его функциями `phash` и `new`.

Другие приемы работы с жесткими ссылками

Как говорилось выше, оператор обратной косой черты обычно используется с одним объектом ссылки для создания одной ссылки, но это не обязательно. В случае применения со списком объектов ссылки он производит список соответствующих ссылок. Вторая строчка в следующем примере делает то же самое, что и первая, поскольку обратная косая черта автоматически распределяется по всему списку.

```
@reflist = (\$s, \@a, \%h, \&f);      # Список из четырех ссылок
@reflist = \($s,  @a   %h,  &f);      # То же самое
```

Если список в круглых скобках содержит ровно один массив или хеш, то интерполируются все их значения и возвращаются ссылки на каждое из них:

```
@reflist = \(@x);                    # Интерполировать массив, затем получить ссылки
@reflist = map { \$_ } @x;            # То же самое
```

То же происходит при наличии внутренних скобок:

```
@reflist = \(@x, (@y));               # Разыменовываются только одиночные составные
                                        объекты
@reflist = (\@x, map { \$_ } @y);     # То же самое
```

Если попробовать проделать это с хешем, то результат будет содержать ссылки на значения (ожидаемый результат) и ссылки на *копии* ключей (довольно неожиданный результат).

Поскольку срезы массивов и хешей представляют собой просто списки, перед любым из них можно поставить обратную косую черту и получить список ссылок. В каждой из трех последующих строк делается одно и то же:

```
@envrefs = \@ENV{'HOME', 'TERM'};         # Обратная косая черта перед срезом
@envrefs = \( $ENV{HOME},  $ENV{TERM} );  # Обратная косая черта перед списком
@envrefs = ( \$ENV{HOME}, \$ENV{TERM} );  # Список из двух ссылок
```

Поскольку функции могут возвращать списки, можно применять к ним обратную косую черту. Если нужно вызвать несколько функций, сначала интерполируйте значения, возвращаемые всеми функциями, в один большой список, а затем поставьте перед ним обратную косую черту:

```
@reflist = \fx();
@reflist = map { \$_ } fx();                # То же самое

@reflist = \( fx(), fy(), fz() );
@reflist = ( \fx(), \fy(), \fz() );         # То же самое
@reflist = map { \$_ } fx(), fy(), fz();    # То же самое
```

Оператор обратной косой черты всегда предоставляет своему операнду списковый контекст, поэтому все эти функции вызываются в списковом контексте. Если сам оператор обратной косой черты находится в скалярном контексте, то в итоге будет получена ссылка на последнее значение списка, возвращенного функцией:

```
@reflist = \localtime();      # Ссылки на все 9 элементов времени
$lastref = \localtime();      # Ссылка на признак летнего времени
```

В этом отношении обратная косая черта ведет себя как именованные списковые операторы Perl, такие как print, reverse и sort, всегда предоставляющие списковый контекст в правой части независимо от того, что происходит в левой. Как и с именованными списковыми операторами, явно используйте scalar, чтобы установить для следующих операндов скалярный контекст:

```
$dateref = \scalar localtime();    # \"Sat Jul 16 11:42:18 2000"
```

Чтобы определить, на что указывает ссылка, можно использовать оператор ref. Рассматривайте ref как оператор «typeof», который возвращает истину, если его аргумент представляет собой ссылку, и ложь в противном случае. Возвращаемое значение зависит от типа объекта ссылки. Встроенными типами являются SCALAR, ARRAY, HASH, CODE, GLOB, REF, LVALUE, IO, IO::Handle и Regexp. Вот как можно проверить аргументы подпрограммы:

```
sub sum {
    my $arrayref = shift;
    warn "Не ссылка на массив" if ref($arrayref) ne "ARRAY";
    return eval join("+", @$arrayref);
}
```

Жесткая ссылка, применяемая в строковом контексте, превращается в строку, содержащую тип и адрес: SCALAR(0x1fc0e). (Обратное преобразование невозможно, поскольку при преобразовании в строку теряются данные о счетчике использования ссылки, а также потому, что было бы опасно позволить программе обращаться к адресу памяти, указываемому в произвольной строке.)

Посредством оператора bless можно устанавливать связь между объектом ссылки и пакетом, действующим как класс объектов. После его выполнения ref возвращает не внутренний тип, а имя класса. При использовании ссылки на объект в строковом контексте возвращается строка, содержащая внешний и внутренний тип, а также адрес памяти: MyType=HASH(0x20d10) или IO::Handle=IO(0x186904). Дополнительные сведения об объектах см. в главе 12 «Объекты».

Поскольку способ, выбранный для разыменовывания, всегда указывает, какого вида объект ссылки нам нужен, typeglob может использоваться так же, как ссылка, несмотря на то, что typeglob содержит несколько объектов ссылки разных типов. Поэтому ${*main::foo} и ${\$main::foo} обращаются к одной и той же скалярной переменной, хотя последнее обращение более эффективно.

Вот прием, с помощью которого можно вставить в строку значение, возвращаемое подпрограммой:

```
print "На этот раз моя подпрограмма возвратила @{[ mysub(1,2,3) ]}.\n";
```

Работает это так. На этапе компиляции, когда обнаруживается `@{...}` в двойных кавычках, это выражение анализируется как блок, возвращающий ссылку. В блоке имеются квадратные скобки, создающие ссылку на анонимный массив, состоящий из того, что находится в скобках. Поэтому на этапе исполнения `mysub(1,2,3)` вызывается в списковом контексте, а результаты помещаются в анонимный массив, ссылка на который возвращается в блоке. Эта ссылка на массив немедленно разыменовывается охватывающим `@{...}`, а значение массива вставляется в строку в двойных кавычках, как это было бы сделано с обычным массивом. Такое крючкотворство удобно использовать и с произвольными выражениями, например:

```
print "Нам нужно @{ [$n + 5] } штуковин!\n";
```

Однако будьте осторожны: квадратные скобки создают списковый контекст для своего выражения. В данном случае это безразлично, но в предыдущем вызове `mysub` могло иметь значение. Когда это существенно, используйте явным образом `scalar` для установления контекста:

```
print "mysub возвращает теперь @{ [scalar mysub(1,2,3)] }.\n";
```

Замыкания

Ранее мы говорили о создании анонимных подпрограмм с помощью безымянного `sub {}`. Можно рассматривать такие подпрограммы как определяемые на этапе исполнения, а это означает, что для них существует момент генерации и должно быть известно местонахождение определения. Во время создания подпрограммы в области видимости могут находиться одни переменные, а при ее вызове – другие переменные.

Забудем на мгновение о подпрограммах и рассмотрим ссылку на лексическую переменную:

```
{
    my $critter = "camel";
    $critterref = \$critter;
}
```

Значением `$$critterref` останется `"camel"`, несмотря на то, что `$critter` исчезает после закрывающей фигурной скобки. Но `$critterref` с таким же успехом могла ссылаться на подпрограмму, ссылающуюся на `$critter`:

```
{
    my $critter = "camel";
    $critterref = sub { return $critter };
}
```

Это и есть *замыкание* (*closure*), понятие которого пришло из области функционального программирования в LISP и Scheme.[1] Оно означает, что если анонимная функция определена в некоторой лексической области видимости в определенный момент, то при вызове этой подпрограммы позднее извне этой области она должна вести себя так, будто находится в этой области видимости. (Пурист сказал бы, что она ничего не должна – она *действительно* выполняется в этой области видимости.)

Иными словами, каждый раз гарантируется получение того же экземпляра лексической переменной, даже если другие экземпляры этой лексической переменной были созданы ранее или позже для других экземпляров этого замыкания. Это дает возможность устанавливать значения, используемые в подпрограмме, во время ее определения, а не только при ее вызове.

Можно также представить себе замыкание как способ написания шаблона подпрограммы без использования `eval`. Лексические переменные выступают в качестве параметров при заполнении шаблона, что удобно для определения небольших фрагментов кода, которые будут выполняться позже. Обычно они называются *функциями обратного вызова* (*callbacks*) в событийно-ориентированном программировании, когда фрагмент кода связывается с нажатием клавиши, щелчком мыши, появлением окна и т. п. Выступая в качестве функций обратного вызова, замыкания делают именно то, чего вы от них ожидаете, даже если у вас нет ни малейшего понятия о функциональном программировании. (Обратите внимание, все, что мы говорили о замыканиях, относится только к лексическим переменным. Глобальные переменные продолжают работать как обычно, потому что они не создаются и не уничтожаются, как это происходит с лексическими переменными.)

Другое применение замыкания находят в *генераторах функций*, т. е. таких функциях, которые создают и возвращают совершенно новые функции. Вот пример генератора функций, реализованного с помощью замыканий:

```
sub make_saying  {
    my $salute = shift;
    my $newfunc = sub {
        my $target = shift;
        print "$salute, $target!\n";
    };
    return $newfunc;            # Возвращает замыкание
}

$f = make_saying("Howdy");      # Создает замыкание
$g = make_saying("Greetings");  # Создает еще одно замыкание

# Прошло некоторое время...

$f->("world");
$g->("earthlings");
```

[1] В данном контексте слово «функция» не должно рассматриваться как антоним к «дисфункция».

В результате выводится:

```
Howdy, world!
Greetings, earthlings!
```

Особенно обратите внимание на то, что $salute продолжает ссылаться на фактическое значение, переданное в make_saying, хотя my $salute вышла из области видимости ко времени выполнения подпрограммы. Для этого и предназначены замыкания. Поскольку $f и $g содержат ссылки на функции, которым при вызове требуется доступ к разным версиям $salute, эти версии автоматически сохраняются. Если теперь перезаписать $f, то *ее* версия $salute автоматически исчезнет. (Perl производит уборку только тогда, когда вы этого не видите.)

Perl не предоставляет ссылки на методы объектов (описываемые в главе 12 «Объекты»), но аналогичных результатов можно добиться с помощью замыкания. Допустим, что нам нужна ссылка не просто на подпрограмму, представляющую метод, но такая, которая может вызывать этот метод с определенным объектом. Легко сохранить объект и метод в качестве лексических переменных, увязанных в замыкании:

```
sub get_method_ref {
    my ($self, $methodname) = @_;
    my $methref = sub {
        # @_ внизу не та же, что вверху!
        return $self->$methodname(@_);
    };
    return $methref;
}

my $dog = new Doggie::          # у меня новая собачонка
            Name => "Lucky",    # по кличке Везунчик,
            Legs => 3,          # у нее три ноги
            Tail => "clipped"; # и куцый хвост.

our $wagger = get_method_ref($dog, 'wag');
$wagger->("tail");         # Вызывает $dog->wag('tail').
                           #
```

Теперь можно заставить Везунчика (Lucky) вилять (wag) тем, что осталось от его хвоста, даже когда лексическая переменная $dog уйдет из области видимости и Lucky не будет видно: глобальная переменная $wagger заставит его выполнить команду wag, где бы он ни находился.

Замыкания как шаблоны функций

Использование замыкания в качестве шаблона функции позволяет генерировать несколько функций, действующих аналогичным образом. Допустим, нам нужен набор функций, генерирующих изменение цвета шрифта в HTML:

```
print "Будь ", red("осторожен "), "с этим ", green("огнем"), "!!!";
```

Функции red и green должны быть очень похожи. Хотелось бы дать им имена, но у замыканий имен нет, т. к. они представляют собой просто анонимные подпрограммы. Чтобы обойти это обстоятельство, прибегнем к хитрому трюку. Можно связать ссылку на код и существующее имя, присвоив его переменной typeglob с именем функции, которая нам нужна. См. раздел «Таблицы имен» главы 10 «Пакеты». В данном случае это будет привязка к двум разным именам, одному на верхнем регистре и другому – на нижнем:

```
@colors = qw(red blue green yellow orange purple violet);
for my $name (@colors) {
    no strict 'refs';       # Разрешить символические ссылки
    *$name = *{uc $name} = sub { "<FONT COLOR='$name'>;@_</FONT>" };
}
```

Теперь можно вызывать функции с именами red, RED, blue, BLUE и т. д.; при этом будет вызываться соответствующее замыкание. Этот прием уменьшает продолжительность компиляции и экономит память, а также уменьшает вероятность ошибок, поскольку проверка синтаксиса происходит во время компиляции. Важно, чтобы при создании замыкания все переменные в анонимной подпрограмме были лексическими. Этим обосновывается появление my в приведенном коде.

Это один из тех редких случаев, когда имеет смысл создать прототип для замыкания. Если требуется установить скалярный контекст для аргументов этих функций (что может быть не очень разумным в данном примере), то можно написать так:

```
*$name = sub ($) { "<FONT COLOR='$name'>$_[0]</FONT>" };
```

Это почти хорошо. Однако, поскольку проверка прототипа происходит на этапе компиляции, вышеприведенное присвоение на этапе исполнения не слишком полезно, т. к. происходит слишком поздно. Положение можно поправить, поместив весь цикл присваиваний в блок BEGIN, чтобы они происходили на этапе компиляции. (Скорее всего, его нужно поместить в модуль, загружаемый use на этапе компиляции.) В этом случае прототипы будут иметь видимость на всем протяжении компиляции.

Вложенные подпрограммы

Если вы привыкли (при работе с другими языками) использовать подпрограммы, которые вкладываются в другие подпрограммы, причем в каждой из них есть собственные закрытые переменные, то в Perl это потребует некоторых дополнительных усилий. Именованные подпрограммы вкладываются одна в другую неправильно, в отличие от анонимных.[1] Тем не менее, мож-

[1] Точнее, глобальные подпрограммы не вкладываются. К сожалению, другого типа объявлений именованных подпрограмм у нас нет. Мы пока не реализовали именованные подпрограммы с лексической областью видимости (известные как my sub), но когда мы это сделаем, они будут вкладываться правильно.

но эмулировать вложенные подпрограммы с лексической видимостью посредством замыканий. Вот пример:

```
sub outer {
    my $x = $_[0] + 35;
    local *inner = sub { return $x * 19 };
    return $x + inner();
}
```

Теперь inner может вызываться только из outer, поскольку в замыкании выполняются временные присвоения. Но если она вызывается, то получает обычный доступ к лексической переменной $x из области видимости outer.

При этом возникает интересный эффект создания функции, локальной для другой функции, что в Perl, вообще говоря, не поддерживается. Поскольку local имеет динамическую область видимости и поскольку имена функций являются глобальными в своих пакетах, любая другая функция, которую вызывает outer, может также вызвать временную версию inner. Чтобы помешать этому, нужно ввести еще один уровень косвенной адресации:

```
sub outer {
    my $x = $_[0] + 35;
    my $inner = sub { return $x * 19 };
    return $x + $inner->();
}
```

Символические ссылки

Что произойдет, если попытаться разыменовать значение, не являющееся жесткой ссылкой? Это значение будет рассматриваться как *символическая ссылка* (*symbolic reference*). Это означает, что ссылка интерпретируется как строка, содержащая *имя* глобальной переменной.

Вот как это действует:

```
$name = "bam";
$$name = 1;                # Устанавливает $bam
$name->[0] = 4;            # Устанавливает первый элемент @bam
$name->{X} = "Y";          # Устанавливает элемент X массива %bam равным Y
@$name = ();               # Очищает @bam
keys %$name;               # Возвращает ключи %bam
&$name;                    # Вызывает &bam
```

Это очень мощное средство и несколько опасное, потому что можно намереваться (совершенно искренне) использовать жесткую ссылку, но случайно указать вместо нее символическую. Чтобы защититься от этого, можно сделать так:

```
use strict 'refs';
```

и тогда до конца охватывающего блока будут разрешены только жесткие ссылки. Вложенный блок может отменить это постановление с помощью

```
no strict 'refs';
```

Важно также понимать различие между следующими двумя строками кода:

```
${identifier};    # То же, что $identifier.
${"identifier"};  # Тоже $identifier, но символическая ссылка.
```

Поскольку во второй форме использованы двойные кавычки, она воспринимается как символическая ссылка, и если действует `use strict 'refs'`, то будет сгенерировано сообщение об ошибке. Даже если `strict 'refs'` не действует, ссылка может осуществляться только на переменную пакета. Первая же форма идентична формату без скобок и будет ссылаться даже на переменную с лексической областью видимости, если таковая объявлена. В следующем примере это демонстрируется (а в следующем разделе обсуждается).

Через символические ссылки доступны только переменные пакета, потому что символические ссылки всегда обрабатываются с помощью таблицы имен пакета. Поскольку лексических переменных нет в таблице имен пакета, они невидимы для данного механизма. Например:

```
our $value  = "global";
{
    my $value = "private";
    print "Inside, mine is ${value}, ";
    print "but ours is ${'value'}.\n";
}
print "Outside, ${value} is again ${'value'}.\n";
```

что выводит:

```
Inside, mine is private, but ours is global.
Outside, global is again global.
[Внутри - мое приватное, а наше - глобальное.
Снаружи глобальное так и остается глобальным.]
```

Фигурные скобки, квадратные скобки и кавычки

В предыдущем разделе мы отметили, что `${identifier}` не считается символической ссылкой. Вас может заинтересовать, как в этом участвуют зарезервированные слова, и кратким ответом будет: «Никак». Несмотря на то что `push` является зарезервированным словом, следующие две команды выводят `"pop on over"`:

```
$push = "pop on ";
print "${push}over";
```

Причина в том, что исторически такое применение фигурных скобок соответствует способу, при помощи которого в оболочках Unix имя переменной отделяется от последующего буквенно-цифрового текста, который, в противном случае, интерпретировался бы как часть имени. Именно такого пове-

дения многие ожидают при интерполяции переменных, поэтому мы осуществили такой же режим в Perl. Но в Perl идея получает дальнейшее развитие и применяется к любым фигурным скобкам, используемым при создании ссылок, независимо от того, заключены ли они в кавычки. Это значит, что:

```
print ${push} . 'over';
```

или даже (поскольку пробелы не имеют значения):

```
print ${ push } . 'over';
```

одинаково выводят "pop on over", несмотря на то, что фигурные скобки находятся вне двойных кавычек. То же правило применимо к любому идентификатору, используемому в качестве индекса хеша. Поэтому вместо:

```
$hash{ "aaa" }{ "bbb" }{ "ccc" }
```

можно просто написать:

```
$hash{ aaa }{ bbb }{ ccc }
```

или:

```
$hash{aaa}{bbb}{ccc}
```

и не беспокоиться о том, что индексы могут быть зарезервированными словами. Поэтому выражение:

```
$hash{ shift }
```

интерпретируется как $hash{"shift"}. Добиться интерпретации как зарезервированного слова можно с помощью добавления чего-либо, лишающего положения просто идентификатора:

```
$hash{  shift()  }
$hash{ +shift   }
$hash{  shift @_ }
```

Ссылки не действуют в качестве ключей хеша

Ключи хешей внутренне хранятся как строки.[1] Если попытаться сохранить ссылку в качестве ключа хеша, значение ключа будет преобразовано в строку:

```
$x{ \$a } = $a;
($key, $value) = each %x;
print $$key;                # НЕВЕРНО
```

Выше говорилось, что нельзя преобразовать строку обратно в жесткую ссылку. Поэтому, если попытаться разыменовать $key, содержащую просто строку,

[1] Они и *внешне* хранятся как строки, например, их помещают в файл DBM. На самом деле, файлы DBM *требуют*, чтобы их ключи (и значения) были строками.

произойдет разыменование не жесткой ссылки, а символической – а т. к., вероятно, переменной с именем SCALAR(0x1fc0e) нет, то попытку завершить не удастся. Скорее надо сделать что-то вроде:

```
$r = \@a;
$x{ $r } = $r;
```

Тогда, по крайней мере, можно будет использовать *значение* хеша, которое будет жесткой ссылкой, а не ключ, который ею не будет.

Хотя нельзя сохранить ссылку как ключ, если (как в вышеприведенном примере) жесткая ссылка используется в строковом контексте, гарантировано, что она создаст уникальную строку, т. к. в результирующую строку включается адрес ссылки. Поэтому на практике можно использовать ссылку в качестве уникального ключа хеша. Просто нельзя будет потом разыменовать ее.

Существует один особый вид хеша, в котором *можно* использовать ссылки в качестве ключей. Благодаря волшебству (magic)[1] модуля Tie::RefHash, поставляемого с Perl, то, что, как мы только что сказали, сделать нельзя, на самом деле сделать можно:

```
use Tie::RefHash;
tie my %h, 'Tie::RefHash';
%h = (
    ["this", "here"]   => "at home",
    ["that", "there"]  => "elsewhere",
);
while ( my($keyref, $value) = each %h ) {
    print "@$keyref is $value\n";
}
```

На самом деле, связывая (tie) различные реализации со встроенными типами, можно добиться от скаляров, хешей и массивов многого из того, чего, как мы говорили, они не могут. Ну, каковы! Бестолковые авторы…

Подробнее о связывании см. главу 14 «Связанные переменные».

Сборка мусора, циклические ссылки и слабые ссылки

Языки высокого уровня обычно позволяют программистам не заботиться об освобождении памяти, которая больше не используется. Этот автоматический процесс переработки отходов известен как *сборка мусора* (*garbage collection*). В большинстве случаев Perl использует быстрый и простой способ сборки мусора, основанный на ссылках.

---

[1] Да, это *действительно* технический термин, что можно заметить, если просмотреть файл *mg.c* в дистрибутиве исходного кода Perl.

При выходе из блока его переменные с локальной областью видимости обычно освобождаются, однако можно так спрятать свой мусор, что сборщик мусора Perl его не найдет. Одна из серьезных опасностей заключается в том, что недоступная память с ненулевым счетчиком ссылок обычно не сможет быть освобождена. Поэтому применение циклических ссылок – плохая идея:

```
{               # заставим $a и $b указывать друг на друга
    my ($a, $b);
    $a = \$b;
    $b = \$a;
}
```

или еще проще:

```
{               # заставим $a указывать на саму себя
    my $a;
    $a = \$a;
}
```

Несмотря на то что `$a` следует освободить в конце блока, это сделано не будет. При создании рекурсивных структур данных необходимо самостоятельно разорвать (или ослабить, см. ниже) автоссылки, если требуется, чтобы память была утилизирована прежде, чем программа (или поток) завершит работу. (При выходе память будет освобождена автоматически путем дорогостоящей, но полной процедуры сборки мусора типа «пометка и очистка» (mark-and-sweep).) Если структура данных является объектом, то для автоматического разрыва ссылки можно применить метод `DESTROY`; см. раздел «Сборка мусора с помощью методов DESTROY» в главе 12 «Объекты».

Аналогичная ситуация может иметь место с *кэшами* – хранилищами данных для ускоренного извлечения. Вне кэша есть ссылки на данные, находящиеся в нем. Проблема возникает, когда все эти ссылки уничтожены, но данные в кэше со своими внутренними ссылками остались. Существование хотя бы одной ссылки не позволяет Perl утилизировать объект ссылки, даже если мы хотим, чтобы данные исчезали из кэша, как только в них отпала нужда. Как и в случае циклических ссылок, нам нужна ссылка, которая не влияет на счетчик ссылок и потому не задерживает сборку мусора.

Слабые ссылки (*weak references*) решают проблемы, вызываемые циклическими ссылками и кэшированными данными, разрешая «ослабить» любую ссылку, т. е. сделать так, чтобы она не влияла на счетчик ссылок. Когда удаляется последняя не слабая ссылка на объект, он уничтожается, и все слабые ссылки на объект автоматически освобождаются.

Для использования этой возможности необходим пакет `WeakRef` из CPAN, в котором содержится дополнительная документация. Слабые ссылки являются экспериментальной функцией. Но кто-то же должен быть подопытным кроликом.

9

Структуры данных

Perl предоставляет многие готовые структуры данных, которые в других языках приходится создавать самостоятельно. Стеки и хеши, изучаемые будущими специалистами по информатике, в Perl являются просто массивами. Когда к массиву применяются `push` и `pop` (или `shift` и `unshift`), это стек; когда к массиву применяются `push` и `shift` (или `unshift` и `pop`), это очередь. А многие из встречающихся древовидных структур создаются только для обеспечения быстрого динамического доступа к таблицам поиска, концептуально являющимся плоскими. Хеши, конечно, встроены в Perl и обеспечивают быстрый динамический доступ к таблицам поиска, концептуально являющимся плоскими, без использования структур данных, от рекурсивности которых цепенеют мозги, и которые кажутся красивыми тем, чьи мозги уже достаточно оцепенели.

Но иногда вложенные структуры данных нужны, потому что они естественным образом моделируют решаемую задачу. Поэтому Perl позволяет комбинировать и вкладывать массивы и хеши для создания структур данных произвольной сложности. При надлежащем применении они могут применяться для создания связанных списков, двоичных деревьев, куч, бинарных деревьев, множеств, графов и всего, что вы еще придумаете. См. книги «Mastering Algorithms with Perl», O'Reilly 1999 (Язык Perl и реализация алгоритмов); «Perl Cookbook», O'Reilly 1998 («Perl: библиотека программиста», СПб, изд. «Питер», 2000) или CPAN – главное хранилище таких модулей. Однако возможно, что вам никогда не понадобится ничего сверх простых сочетаний массивов и хешей, поэтому о них-то мы и поговорим в этой главе.

Массивы массивов

Есть много видов вложенных структур данных. Проще всего построить массив массивов, называемый также двумерным массивом или матрицей. (Допустимо очевидное обобщение: массив массивов массивов является трехмерным массивом, и т. д. для большего числа измерений.) Разобраться с ним достаточно просто, и почти все, что применимо в данном случае, может применяться к более замысловатым структурам данных, которые мы будем изучать в следующих разделах.

Создание двумерного массива и доступ к нему

Вот как собирается двумерный массив:

```
#Присвоить массиву список ссылок на массивы.
@AoA = (
         [ "fred", "barney" ],
         [ "george", "jane", "elroy" ],
         [ "homer", "marge", "bart" ],
);

print $AoA[2][1];   # выводит "marge"
```

Весь список заключен в круглые, а не в квадратные скобки, поскольку значения присваиваются списку, а не ссылке. Для того чтобы получить ссылку на массив, следует использовать квадратные скобки:

```
# Создать ссылку на массив ссылок на массивы.
$ref_to_AoA = [
    [ "fred", "barney", "pebbles", "bamm bamm", "dino", ],
    [ "homer", "bart", "marge", "maggie", ],
    [ "george", "jane", "elroy", "judy", ],
];

print $ref_to_AoA->[2][3];   # выводит "judy"
```

Помните, что между каждой парой примыкающих фигурных или квадратных скобок подразумевается присутствие `->`. Поэтому следующие две строки:

```
$AoA[2][3]
$ref_to_AoA->[2][3]
```

эквивалентны таким двум строкам:

```
$AoA[2]->[3]
$ref_to_AoA->[2]->[3]
```

Здесь, однако, перед первой парой квадратных скобок не подразумевается присутствия `->`, но для разыменования `$ref_to_AoA` начальный `->` требуется. Помните также, что можно вести обратный отсчет от конца массива с помощью отрицательного индекса, поэтому:

```
$AoA[0][-2]
```

является предпоследним элементом первой строки.

Наращивание массива

Такие присвоения большим спискам годятся для создания фиксированных структур данных, но как быть, если нужно вычислять каждый элемент на лету или иным образом строить структуру по частям?

Прочтем структуру данных из файла. Будем предполагать, что это плоский текстовый файл, в котором каждая строчка является строкой данных и состоит из элементов, разделяемых пробельными символами. Вот как нужно действовать:[1]

```
while (<>) {
    @tmp = split;           # Расщепить элементы в массив.
    push @AoA, [ @tmp ];    # Добавить к @AoA ссылку на анонимный массив.
}
```

Конечно, нет необходимости давать имя временному массиву, поэтому можно сказать и так:

```
while (<>) {
    push @AoA, [ split ];
}
```

Если нужна ссылка на массив массивов, можно сделать так:

```
while (<>) {
    push @$ref_to_AoA, [ split ];
}
```

В обоих этих примерах к массиву массивов добавляются новые строки. А как добавить новые колонки? При работе с двумерными массивами часто проще всего использовать обычное присваивание:[2]

```
for $x (0 .. 9) {                       # Для каждой строки...
    for $y (0 .. 9) {                   # Для каждой колонки...
        $AoA[$x][$y] = func($x, $y);    # ...присвоить этой ячейке
    }
}

for $x ( 0..9 ) {                       # Для каждой строки...
    $ref_to_AoA->[$x][3] = func2($x);   # ...установить четвертую колонку
}
```

---

[1] Здесь, как и в других главах, мы опускаем (для ясности) обычно помещаемые объявления `my`. В данном примере надо было бы написать `my @tmp = split`.

[2] Как и в вышеприведенном присваивании временной переменной, мы произвели упрощение: в действительном коде для циклов, обсуждаемых в этой главе, следовало бы, вероятно, записывать `for my $x`.

Порядок присваивания элементам значений не важен, как и то, существуют ли уже элементы @AoA с указываемыми индексами: Perl охотно создаст их для вас, установив, как водится, промежуточные элементы равными неопределенному значению. (Perl даже создаст, при необходимости, первую ссылку в $ref_to_AoA.) Для добавления строки нужно сделать нечто более замысловатое:

```
# Добавить новые колонки к существующей строке.
push @{ $AoA[0] }, "wilma", "betty";
```

Обратите внимание, что следующая команда не будет работать:

```
push $AoA[0], "wilma", "betty";  # НЕВЕРНО!
```

Это даже не будет скомпилировано, потому что аргумент команды push должен быть настоящим массивом, а не ссылкой на массив. Поэтому первый аргумент обязательно должен начинаться с символа @. То, что следует за @, может быть предметом обсуждения.

Доступ к массивам и вывод

А теперь выведем структуру данных. Если нужен только один элемент, достаточно следующего:

```
print $AoA[3][2];
```

Но чтобы напечатать весь массив, нельзя просто сказать:

```
print @AoA;         # НЕВЕРНО
```

Это неправильно, и вместо своих данных вы увидите ссылки, преобразованные в строки. Perl не выполняет разыменование автоматически, поэтому придется написать один-два цикла. Следующий код выводит всю структуру, проходя в цикле элементы @AoA и разыменовывая каждый из них в команде print:

```
for $row ( @AoA ) {
    print "@$row\n";
}
```

Если вы хотите следить за индексами, то можно сделать так:

```
for $i ( 0 .. $#AoA ) {
    print "строка $i: @{$AoA[$i]}\n";
}
```

или даже так (обратите внимание на внутренний цикл):

```
for $i ( 0 .. $#AoA ) {
    for $j ( 0 .. $#{$AoA[$i]} ) {
        print "элемент $i $j : $AoA[$i][$j]\n";
    }
}
```

Как видите, дело немного усложняется. Поэтому иногда проще решить задачу с помощью временной переменной:

```
for $i ( 0 .. $#AoA ) {
    $row = $AoA[$i];
    for $j ( 0 .. $#{$row} ) {
        print "элемент $i $j : $row->[$j]\n";
    }
}
```

Срезы

Если нужно получить доступ к срезу (части строки) многомерного массива, придется использовать индексы несколько необычным способом. Стрелки указателей обеспечивают удобный доступ к отдельному элементу, но для срезов таких удобств не существует. Элементы среза всегда можно получить один за другим с помощью цикла:

```
@part = ();
for ($y = 7; $y < 13; $y++) {
    push @part, $AoA[4][$y];
}
```

Данный цикл можно заменить срезом массива:

```
@part = @{ $AoA[4] } [ 7..12 ];
```

Если требуется *двумерный срез*, например, для `$x` в диапазоне `4..8` и `$y` в диапазоне `7..12`, то его можно получить так:

```
@newAoA = ();
for ($startx = $x = 4; $x <= 8; $x++) {
    for ($starty = $y = 7; $y <= 12; $y++) {
        $newAoA[$x - $startx][$y - $starty] = $AoA[$x][$y];
    }
}
```

В этом примере отдельные значения нашего целевого двумерного массива `@newAoA` присваиваются одно за другим, взятые из двумерного подмассива `@AoA`. Другой возможностью является создание анонимных массивов, каждый из которых состоит из нужного среза подмассива `@AoA`, и помещение ссылок на эти анонимные массивы в `@newAoA`. В этом случае мы будем записывать ссылки в `@newAoA` (так сказать, один раз проиндексированный), а не значения подмассива в дважды проиндексированный `@newAoA`. При таком методе устраняется самый внутренний цикл:

```
for ($x = 4; $x <= 8; $x++) {
    push @newAoA, [ @{ $AoA[$x] } [ 7..12 ] ];
}
```

Конечно, если вы делаете это часто, то, вероятно, следует написать подпрограмму, назвав ее, например, extract_rectangle. А если вы делаете это очень часто и с большими наборами многомерных данных, то следует, вероятно, воспользоваться модулем PDL (Perl Data Language), который можно получить из CPAN.

Распространенные ошибки

Как говорилось ранее, массивы и хеши в Perl являются одномерными. Даже «многомерные» массивы фактически имеют в Perl одно измерение, но значения в них являются ссылками на другие массивы, сжимающими несколько элементов в один. При выводе этих значений без разыменования будут получены не требуемые данные, а ссылки, преобразованные в строки. Например, следующие две строки:

```
@AoA = ( [2, 3], [4, 5, 7], [0] );
print "@AoA";
```

выводят нечто типа:

```
ARRAY(0x83c38) ARRAY(0x8b194) ARRAY(0x8b1d0)
```

С другой стороны, следующая строка выводит 7:

```
print $AoA[1][2];
```

При построении массива массивов не забывайте создавать новые ссылки для подмассивов, иначе получится массив, элементы которого будут представлять число элементов в подмассивах, например:

```
for $i (1..10) {
    @array = somefunc($i);
    $AoA[$i] = @array;       # НЕВЕРНО!
}
```

Здесь доступ к @array осуществляется в скалярном контексте и поэтому дает число его элементов, которое послушно присваивается элементу $AoA[$i]. Как правильно присваивать ссылку, будет показано через мгновение.

Сделав такую ошибку, человек понимает, что нужно присвоить ссылку, поэтому следующая естественная ошибка – это многократное получение ссылки на один и тот же адрес:

```
for $i (1..10) {
    @array = somefunc($i);
    $AoA[$i] = \@array;      # СНОВА НЕВЕРНО!
}
```

Все ссылки, создаваемые во второй строке цикла, будут одинаковыми, а именно ссылками на единственный массив @array. Да, этот массив изменяется при каждом проходе цикла, но когда все закончено, $AoA содержит

10 ссылок на один и тот же массив, в котором содержится последний набор присвоенных ему значений. Вывод `@{$AoA[1]}` показывает те же значения, что и вывод `@{$AoA[2]}`.

Вот более удачный подход:

```
for $i (1..10) {
    @array = somefunc($i);
    $AoA[$i] = [ @array ];   # ПРАВИЛЬНО!
}
```

Квадратные скобки вокруг массива `@array` создают новый анонимный массив, в который копируются значения элементов `@array`. Затем мы запоминаем ссылку на этот новый массив.

Аналогичный результат производится кодом (более трудно читаемым):

```
for $i (1..10) {
    @array = somefunc($i);
    @{$AoA[$i]} = @array;
}
```

Поскольку элемент `$AoA[$i]` должен быть новой ссылкой, эта ссылка создается. Затем предшествующий `@` разыменовывает эту новую ссылку, в результате чего значения `@array` присваиваются (в списковом контексте) массиву, на который ссылается `$AoA[$i]`. Возможно, чтобы код был более понятным, такой конструкции следует избегать.

Однако *есть* ситуация, в которой можно было бы ее использовать. Допустим, что `@AoA` уже является массивом ссылок на массивы, т. е. выполнены присваивания типа:

```
$AoA[3] = \@original_array;
```

А теперь предположим, что нужно изменить `@original_array` (т. е. четвертую строку `$AoA`) так, чтобы он ссылался на элементы `@array`. Для этого подойдет такой код:

```
@{$AoA[3]} = @array;
```

В данном случае сама ссылка не меняется, изменяются элементы массива, на который производится ссылка. В результате перезаписываются значения массива `@original_array`.

Наконец, следующий страшно выглядящий код прекрасно работает:

```
for $i (1..10) {
    my @array = somefunc($i);
    $AoA[$i] = \@array;
}
```

Это происходит благодаря тому, что переменная с лексической областью видимости `my @array` создается заново при каждом проходе цикла. Поэтому,

несмотря на то, что кажется, будто каждый раз сохраняется ссылка на одну и ту же переменную, на самом деле это не так. Это тонкое различие, но такой прием создает более эффективный код, хотя при этом возникает опасность ввести в заблуждение не слишком опытных программистов. (Он более эффективен, поскольку в последнем присваивании не осуществляется копирование.) С другой стороны, если все равно приходится копировать значения (что осуществляет первое присваивание в цикле), то можно так же использовать копирование, которое вызывается квадратными скобками, и избавиться от временной переменной:

```
for $i (1..10) {
    $AoA[$i] = [ somefunc($i) ];
}
```

Подведем итоги:

```
$AoA[$i] = [ @array ];   # Самое безопасное, иногда самое быстрое
$AoA[$i] = \@array;      # Быстро, но рискованно, зависит от my массива
@{ $AoA[$i] } = @array;  # Несколько мудрено
```

Овладев применением массивов, состоящих из массивов, вы захотите заняться более сложными структурами данных. Тот, кто ищет структуры C или записи Pascal, не найдет в Perl зарезервированных слов, которые их создают. Вместо них предоставляется более гибкая система. Если вы считаете, что структура записи должна быть менее гибкой, или хотите дать своим пользователям нечто менее прозрачное и более жесткое, то можете воспользоваться объектно-ориентированными функциями, подробно описываемыми в главе 12 «Объекты».

В Perl всего два способа организации данных: в виде упорядоченных списков, хранимых в массивах, доступ к которым осуществляется по расположению, и в виде неупорядоченных пар ключ/значение, хранимых в хешах, доступ к которым осуществляется по имени. В Perl лучше всего представлять записи как ссылки на хеш, но выбор способа организации таких записей может быть различным. Возможно, кому-то понадобится хранить упорядоченный список этих записей, и тогда следует использовать для хранения записей массив ссылок на хеши. Либо может потребоваться поиск записей по именам, и тогда следует создать хеш ссылок на хеш. Можно даже сделать то и другое одновременно с помощью псевдохеша.

В следующих разделах приведены примеры кода, подробно показывающие, как формировать (с самого начала), генерировать (на основании других источников), получать доступ и отображать несколько различных типов структур данных. Сначала мы демонстрируем три простые комбинации массивов и хешей, затем хеш функций и более необычные структуры данных. Закончим мы демонстрацией того, как можно сохранять эти структуры данных. В этих примерах предполагается, что вы хорошо освоили материал, изложенный ранее в этой главе.

Хеши массивов

Используйте хеш массивов, когда нужно находить каждый массив по некоторой строке, а не просто по номеру индекса. В нашем примере с персонажами телесериалов вместо того, чтобы просматривать списки имен 0-й передачи, 1-й передачи и т. д., мы сделаем так, что можно будет найти состав действующих лиц по названию передачи.

Поскольку внешней структурой данных у нас является хеш, мы не можем упорядочить содержимое, но можем использовать функцию `sort` для задания типа упорядочения выводимых данных.

Формирование хеша массивов

Хеш анонимных массивов можно создать так:

```
# Обычно мы опускаем кавычки, когда ключи являются идентификаторами.
%HoA = (
    flintstones    => [ "fred", "barney" ],
    jetsons        => [ "george", "jane", "elroy" ],
    simpsons       => [ "homer", "marge", "bart" ],
);
```

Чтобы добавить к хешу еще один массив, можно просто сказать:

```
$HoA{teletubbies} = [ "tinky winky", "dipsy", "laa-laa", "po" ];
```

Генерирование хеша массивов

Вот несколько приемов, с помощью которых можно заполнить хеш массивов. Чтобы произвести чтение из массива следующего формата:

```
flintstones: fred barney wilma dino
jetsons:     george jane elroy
simpsons:    homer marge bart
```

можно использовать один из следующих циклов:

```
while ( <> ) {
    next unless s/^(.*?):\s*//;
    $HoA{$1} = [ split ];
}

while ( $line = <> ) {
    ($who, $rest) = split /:\s*/, $line, 2;
    @fields = split ' ', $rest;
    $HoA{$who} = [ @fields ];
}
```

Если имеется подпрограмма `get_family`, которая возвращает массив, можно использовать ее для заполнения `%HoA` с помощью одного из двух следующих циклов:

```
for $group ( "simpsons", "jetsons", "flintstones" ) {
    $HoA{$group} = [ get_family($group) ];
}

for $group ( "simpsons", "jetsons", "flintstones" ) {
    @members = get_family($group);
    $HoA{$group} = [ @members ];
}
```

Новые члены можно дописать к существующему массиву следующим образом:

```
push @{ $HoA{flintstones} }, "wilma", "pebbles";
```

Доступ к хешу массивов и вывод его элементов

Присвоить значение первому элементу некоторого массива можно так:

```
$HoA{flintstones}[0] = "Fred";
```

Для записи с заглавной буквы второго из Симпсонов примените подстановку к соответствующему элементу массива:

```
$HoA{simpsons}[1] =~ s/(\w)/\u$1/;
```

Все семейства можно вывести, пройдя в цикле все ключи хеша:

```
for $family ( keys %HoA ) {
    print "$family: @{ $HoA{$family} }\n";
}
```

Приложив некоторые дополнительные усилия, можно добавить и индексы массивов:

```
for $family ( keys %HoA ) {
    print "$family: ";
    for $i ( 0 .. $#{ $HoA{$family} } ) {
        print " $i = $HoA{$family}[$i]";
    }
    print "\n";
}
```

Или отсортировать массивы по числу входящих в них элементов:

```
for $family ( sort { @{$HoA{$b}} <=> @{$HoA{$a}} } keys %HoA ) {
    print "$family: @{ $HoA{$family} }\n"
}
```

Или даже отсортировать массивы по числу элементов, а затем упорядочить массивы в соответствии с таблицей ASCII (или, более точно, utf8):

```
# Вывести все, упорядочив по числу элементов и фамилиям.
for $family ( sort { @{$HoA{$b}} <=> @{$HoA{$a}} } keys %HoA ) {
```

```
    print "$family: ", join(", ", sort @{ $HoA{$family} }), "\n";
}
```

Массивы хешей

Массив хешей полезен, когда есть группа записей, к которым нужно обращаться последовательно, а каждая запись содержит пары ключ/значение. Массивы хешей используются реже, чем прочие структуры, представленные в данной главе.

Формирование массива хешей

Массив анонимных хешей можно создать так:

```
@AoH = (
    {
       husband  => "barney",
       wife     => "betty",
       son      => "bamm bamm",
    },
    {
       husband => "george",
       wife    => "jane",
       son     => "elroy",
    },
       husband => "homer",
       wife    => "marge",
       son     => "bart",
    },
  );
```

Для добавления к массиву еще одного хеша можно просто сказать:

```
push @AoH, { husband => "fred", wife => "wilma", daughter => "pebbles" };
```

Генерирование массива хешей

Вот несколько приемов, с помощью которых можно заполнить массив хешей. Чтобы произвести чтение из массива такого формата:

```
husband=fred friend=barney
```

можно использовать один из следующих циклов:

```
while ( <> ) {
    $rec = {};
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
        $rec->{$key} = $value;
    }
```

```
    push @AoH, $rec;
}

while ( <> ) {
    push @AoH, { split /[\s=]+/ };
}
```

Если имеется подпрограмма get_next_pair, которая возвращает пары ключ/значение, можно применить ее для заполнения @AoH с помощью одного из двух следующих циклов:

```
while ( @fields = get_next_pair() ) {
    push @AoH, { @fields };
}

while (<>) {
    push @AoH, { get_next_pair($_) };
}
```

Новые члены можно дописать к существующему хешу так:

```
$AoH[0]{pet} = "dino";
$AoH[2]{pet} = "santa's little helper";
```

Доступ к массиву хешей и вывод его элементов

Задать пару ключ/значение в некотором хеше можно так:

```
$AoH[0]{husband} = "fred";
```

Для того чтобы значение husband из второго массива начиналось с прописной буквы, примените подстановку:

```
$AoH[1]{husband} =~ s/(\w)/\u$1/;
```

Вывести все данные можно так:

```
for $href ( @AoH ) {
    print "{ ";
    for $role ( keys %$href ) {
         print "$role=$href->{$role} ";
    }
    print "}\n";
}
```

и с индексами:

```
for $i ( 0 .. $#AoH ) {
    print "$i is { ";
    for $role ( keys %{ $AoH[$i] } ) {
         print "$role=$AoH[$i]{$role} ";
    }
    print "}\n";
}
```

Хеши хешей

Многомерный хеш является самой гибкой из вложенных структур Perl. Он напоминает запись, которая, в свою очередь, содержит другие записи. На каждом уровне индекс хеша задается с помощью строки (при необходимости заключаемой в кавычки). Не забывайте, однако, что пары ключ/значение извлекаются из хеша в произвольном порядке. Для извлечения пар в нужном порядке можно использовать функцию `sort`.

Формирование хеша хешей

Хеш анонимных хешей можно создать так:

```
%HoH = (
    flintstones => {
        husband   => "fred",
        pal       => "barney",
    },
    jetsons => {
        husband   => "george",
        wife      => "jane",
        "his boy" => "elroy",  # Ключи должны быть в кавычках.
    },
    simpsons => {
        husband   => "homer",
        wife      => "marge",
        kid       => "bart",
    },
);
```

Чтобы добавить к `%HoH` еще один анонимный хеш, можно просто сказать:

```
$HoH{ mash } = {
    captain  => "pierce",
    major    => "burns",
    corporal => "radar",
};
```

Генерирование хеша хешей

Вот несколько приемов, с помощью которых можно заполнить хеш хешей. Чтобы произвести чтение из массива такого формата:

```
flintstones: husband=fred pal=barney wife=wilma pet=dino
```

можно использовать один из следующих циклов:

```
while ( <> ) {
    next unless s/^(.*?):\s*//;
    $who = $1;
    for $field ( split ) {
```

```
        ($key, $value) = split /=/, $field;
        $HoH{$who}{$key} = $value;
    }
}

while ( <> ) {
    next unless s/^(.*?):\s*//;
    $who = $1;
    $rec = {};
    $HoH{$who} = $rec;
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
        $rec->{$key} = $value;
    }
}
```

Если имеется подпрограмма `get_family`, которая возвращает список пар ключ/значение, можно использовать ее для заполнения `%HoH` с помощью одного из трех следующих фрагментов кода:

```
for $group ( "simpsons", "jetsons", "flintstones" ) {
    $HoH{$group} = { get_family($group) };
}

for $group ( "simpsons", "jetsons", "flintstones" ) {
    @members = get_family($group);
    $HoH{$group} = { @members };
}
sub hash_families {
    my @ret;
    for $group ( @_ ) {
        push @ret, $group, { get_family($group) };
    }
    return @ret;
}
%HoH = hash_families( "simpsons", "jetsons", "flintstones" );
```

Новые члены можно дописать к существующему хешу так:

```
%new_folks = (
    wife => "wilma",
    pet  => "dino";
);
for $what (keys %new_folks) {
    $HoH{flintstones}{$what} = $new_folks{$what};
}
```

Доступ к хешу хешей и вывод его элементов

Установить значение пары ключ/значение в некотором хеше можно так:

```
$HoH{flintstones}{wife} = "wilma";
```

Для записи с заглавной буквы некоторой пары ключ/значение примените к элементу подстановку:

```
$HoH{jetsons}{'his boy'} =~ s/(\w)/\u$1/;
```

Состав всех семейств можно вывести путем перебора в цикле ключей внешнего хеша, а затем – перебора в цикле ключей внутреннего хеша:

```
for $family ( keys %HoH ) {
    print "$family: ";
    for $role ( keys %{ $HoH{$family} } ) {
         print "$role=$HoH{$family}{$role} ";
    }
    print "\n";
}
```

Для очень больших хешей несколько более быстрым способом может оказаться одновременное извлечение ключей и значений с помощью each (что помешает сортировке):

```
while ( ($family, $roles) = each %HoH ) {
    print "$family: ";
    while ( ($role, $person) = each %$roles ) {
        print "$role=$person ";
    }
    print "\n";
}
```

(К несчастью, как раз большие хеши и приходится сортировать, в противном случае в выведенном невозможно найти то, что требуется.) Можно отсортировать семейства, а затем роли следующим образом:

```
for $family ( sort keys %HoH ) {
    print "$family: ";
    for $role ( sort keys %{ $HoH{$family} } ) {
         print "$role=$HoH{$family}{$role} ";
    }
    print "\n";
}
```

Чтобы отсортировать семейства по числу членов (а не по ASCII (или utf8)), можно использовать keys в скалярном контексте:

```
for $family ( sort { keys %{$HoH{$a}} <=> keys %{$HoH{$b}} } keys %HoH ) {
    print "$family: ";
    for $role ( sort keys %{ $HoH{$family} } ) {
         print "$role=$HoH{$family}{$role} ";
    }
    print "\n";
}
```

Для сортировки членов семейств в некотором фиксированном порядке можно присвоить им ранги:

```
$i = 0;
for ( qw(husband wife son daughter pal pet) ) { $rank{$_} = ++$i }

for $family ( sort { keys %{$HoH{$a}} <=> keys %{$HoH{$b}} } keys %HoH ) {
    print "$family: ";
    for $role ( sort { $rank{$a} <=> $rank{$b} } keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "\n";
}
```

Хеши функций

Когда на Perl пишут сложное приложение или сетевой сервис, может потребоваться сделать доступными для пользователей большое число команд. В такой программе для определения выбора пользователя и выполнения соответствующих действий может иметься код вроде следующего:

```
if    ($cmd =~ /^exit$/i)     { exit }
elsif ($cmd =~ /^help$/i)     { show_help() }
elsif ($cmd =~ /^watch$/i)    { $watch = 1 }
elsif ($cmd =~ /^mail$/i)     { mail_msg($msg) }
elsif ($cmd =~ /^edit$/i)     { $edited++; editmsg($msg); }
elsif ($cmd =~ /^delete$/i)   { confirm_kill() }
else {
    warn "Неизвестная команда: `$cmd'; попробуйте вызвать`help'\n";
}
```

В структурах данных можно запоминать ссылки на функции точно так же, как ссылки на массивы и хеши:

```
%HoF = (                            # Формирование хеша функций
    exit    =>  sub { exit },
    help    =>  \&show_help,
    watch   =>  sub { $watch = 1 },
    mail    =>  sub { mail_msg($msg) },
    edit    =>  sub { $edited++; editmsg($msg); },
    delete  =>  \&confirm_kill,
);

if   ($HoF{lc $cmd}) { $HoF{lc $cmd}->() }   # Вызов функции
else { warn " Неизвестная команда: `$cmd'; попробуйте вызвать`help'\n" }
```

В предпоследней строке проверяется, существует ли заданное имя команды (на нижнем регистре) в «таблице рассылки», `%HoF`. Если да, то мы вызываем соответствующую команду, разыменовывая значение хеша как функцию и передавая этой функции пустой список аргументов. Мы могли бы разыменовать ее также как `&{ $HoF{lc $cmd} }()` или, в версии Perl 5.6, просто `$HoF{lc $cmd}()`.

Более сложные записи

То, что мы до сих пор видели в этой главе, представляет собой простые двухуровневые однородные структуры данных: каждый элемент содержит объекты ссылки того же типа, что и остальные элементы на том же уровне. Так, конечно, должно быть не обязательно. Любой элемент может содержать скаляр любого типа, т. е. может быть строкой, числом или ссылкой на что угодно. Ссылка может быть ссылкой на массив или хеш, псевдохешем, ссылкой на именованную или анонимную функцию или объект. Единственное, чего нельзя делать, это помещать в один скаляр несколько объектов ссылки. Если вы обнаружите, что делаете такую попытку, значит, вам нужна ссылка на массив или хеш, чтобы сжать несколько значений в одно.

В последующих разделах вы найдете примеры кода, иллюстрирующие хранение многих возможных типов данных в записях, реализуемых с помощью ссылок на хеш. Ключами являются строки, набранные в верхнем регистре, – соглашение, иногда применяемое, когда хеш выступает в качестве особого типа записи.

Формирование, доступ и вывод более сложных записей

Вот запись, содержащая шесть в корне отличающихся по типу полей:

```
$rec = {
    TEXT      => $string,
    SEQUENCE  => [ @old_values ],
    LOOKUP    => { %some_table },
    THATCODE  => \&some_function,
    THISCODE  => sub { $_[0] ** $_[1] },
    HANDLE    => \*STDOUT,
};
```

Поле TEXT является обыкновенной строкой, поэтому его можно вывести:

```
print $rec->{TEXT};
```

SEQUENCE и LOOKUP представляют собой обычные ссылки на массив и хеш:

```
print $rec->{SEQUENCE}[0];
$last = pop @{ $rec->{SEQUENCE} };
print $rec->{LOOKUP}{"key"};
($first_k, $first_v) = each %{ $rec->{LOOKUP} };
```

THATCODE – это именованная подпрограмма, а THISCODE – анонимная подпрограмма, но вызываются они идентичным образом:

```
$that_answer = $rec->{THATCODE}->($arg1, $arg2);
$this_answer = $rec->{THISCODE}->($arg1, $arg2);
```

С помощью дополнительной пары фигурных скобок можно обращаться с `$rec->{HANDLE}` как с косвенным объектом:

```
print { $rec->{HANDLE} } "строка\n";
```

А применение модуля `FileHandle` позволяет даже обращаться с дескриптором как с обычным объектом:

```
use FileHandle;
$rec->{HANDLE}->autoflush(1);
$rec->{HANDLE}->print("строка\n");
```

Формирование, доступ и вывод еще более сложных записей

Естественно, что поля ваших структур данных «имеют право» сами быть структурами данных произвольной сложности:

```
%TV = (
    flintstones => {
        series   => "flintstones",
        nights   => [ "monday", "thursday", "friday" ],
        members  => [
            { name => "fred",    role => "husband", age  => 36, },
            { name => "wilma",   role => "wife",    age  => 31, },
            { name => "pebbles", role => "kid",     age  =>  4, },
        ],
    },
        series   => "jetsons",
        nights   => [ "wednesday", "saturday" ],
        members  => [
            { name => "george",  role => "husband", age  => 41, },
            { name => "jane",    role => "wife",    age  => 39, },
            { name => "elroy",   role => "kid",     age  =>  9, },
        ],
    },
    simpsons    => {
        series   => "simpsons",
        nights   => [ "monday" ],
        members  => [
            { name => "homer", role => "husband", age => 34, },
            { name => "marge", role => "wife",    age => 37, },
            { name => "bart",  role => "kid",     age => 11, },
        ],
    },
);
```

Генерирование хеша сложных записей

Поскольку Perl хорошо анализирует сложные структуры данных, можно поместить объявления данных в отдельный файл в виде обычного кода Perl,

а затем загружать их с помощью встроенных функций do или require. Другим распространенным подходом является применение модуля CPAN (такого, как XML::Parser) для загрузки произвольных структур данных, описанных на каком-то другом языке (например, XML).

Можно строить структуры данных по частям:

```
$rec = {};
$rec->{series} = "flintstones";
$rec->{nights} = [ find_days() ];
```

Или считывать их из файла (в данном случае предполагается использование синтаксиса поле=значение):

```
@members = ();
while (<>) {
     %fields = split /[\s=]+/;
     push @members, { %fields };
}
$rec->{members} = [ @members ];
```

И вкладывать их в более крупные структуры данных, используя в качестве ключа одно из подполей:

```
$TV{ $rec->{series} } = $rec;
```

Применение дополнительных полей с указателями позволяет избежать дублирования данных. Например, в запись о человеке можно включить поле "kids", которое может быть ссылкой на массив, содержащий ссылки на собственные записи, относящиеся к детям. Заставив одни части структуры данных ссылаться на другие, можно избежать противоречивости данных, возникающей при их обновлении лишь в одном месте из нескольких необходимых:

```
for $family (keys %TV) {
    my $rec = $TV{$family};   # временный указатель
    @kids = ();
    for $person ( @{$rec->{members}} ) {
        if ($person->{role} =~ /kid|son|daughter/) {
            push @kids, $person;
        }
    }
    # $rec и $TV{$family} указывают на одни и те же данные!
    $rec->{kids} = [ @kids ];
}
```

Присваивание $rec->{kids} = [@kids] копирует содержимое массива, которое представляет собой просто ссылки на некопируемые данные. Это значит, что если увеличить возраст Барта:

```
$TV{simpsons}{kids}[0]{age}++;            # увеличивает до 12
```

то, поскольку $TV{simpsons}{kids}[0] и $TV{simpsons}{members}[2] указывают на одну и ту же анонимную хеш-таблицу, мы увидим следующий результат:

```
print $TV{simpsons}{members}[2]{age};     # тоже выводит 12
```

Теперь выведем всю структуру %TV:

```
for $family ( keys %TV ) {
    print "the $family";
    print " is on ", join (" and ", @{ $TV{$family}{nights} }), "\n";
    print "its members are:\n";
    for $who ( @{ $TV{$family}{members} } ) {
        print " $who->{name} ($who->{role}), age $who->{age}\n";
    }
    print "children: ";
    print join (", ", map { $_->{name} } @{ $TV{$family}{kids} } );
    print "\n\n";
}
```

Сохранение структур данных

Если вы хотите сохранить свои структуры данных для использования в дальнейшем другой программой, то сделать это можно несколькими способами. Проще всего применить модуль Perl Data::Dumper, который преобразует структуру данных (возможно, содержащую ссылки на саму себя) в строку, которую можно сохранить во внешней памяти, а затем воспроизвести с помощью eval или do.

```
use Data::Dumper;
$Data::Dumper::Purity = 1;       # поскольку %TV содержит ссылки на себя
open (FILE, "> tvinfo.perldata") or die "невозможно открыть tvinfo: $!";
print FILE Data::Dumper->Dump([\%TV], ['*TV']);
close FILE                       or die "невозможно закрыть tvinfo: $!";
```

Отдельная программа (или та же самая) может позже считать этот файл:

```
open (FILE, "< tvinfo.perldata") or die "невозможно открыть tvinfo: $!";
undef $/;                        # считать сразу весь файл
eval <FILE>;                     # воссоздать %TV
die "невозможно воссоздать данные tv из tvinfo.perldata: $@" if $@;
close FILE                       or die "невозможно закрыть tvinfo: $!";
print $TV{simpsons}{members}[2]{age};
```

или просто:

```
do "tvinfo.perldata"            or die " невозможно воссоздать tvinfo: $! $@";
print $TV{simpsons}{members}[2]{age};
```

Есть много других решений, в которых формат записи изменяется от упакованного двоичного (очень быстрого) до XML (очень совместимого). Сегодня же посмотрите на ближайшем зеркале CPAN!

10

Пакеты

В этой главе мы добрались до приятных вещей, поскольку речь пойдет о разработке программного обеспечения. Если мы собираемся обсуждать правильное проектирование программного обеспечения, то должны поговорить о Лени, Нетерпении и Самодовольстве – основах правильной разработки программного обеспечения.

Мы все попадали в ловушку копирования через буфер (cut-and-paste), когда следовало определить абстракцию более высокого уровня, даже если это были лишь цикл или подпрограмма.[1] Несомненно, кое-кто впадал в противоположную крайность, нагромождая кучи абстракций более высокого уровня, когда надо было лишь скопировать код через буфер.[2] Все же большинству из нас следует подумать о том, чтобы использовать абстракции почаще.

Промежуточное положение занимают приверженцы взвешенного подхода к применению абстракций, которые, однако, слишком рано бросаются создавать собственные абстракции, когда следовало бы повторно использовать имеющийся код.[3]

Испытав соблазн сделать что-либо из перечисленного, откиньтесь в кресле и подумайте, что в конечном итоге окажется лучше всего для вас и вашего ближнего. Если вы хотите излить свою творческую энергию в программе, почему бы при этом не сделать мир совершеннее? (Даже если вашей един-

---

[1] Это разновидность Ложной Лени.

[2] Это разновидность Ложного Самомнения.

[3] Вы угадали: это Ложное Нетерпение. Но если вы намерены изобретать заново колесо, постарайтесь хотя бы, чтобы ваше колесо было более удачным.

ственной целью является *успех* программы, вы должны убедиться, что она займет нужную экологическую нишу.)

Первый шаг на пути к экологически чистому программированию очень прост: не надо мусорить в парке. Когда вы пишете некоторый код, подумайте о создании для него собственного пространства имен, чтобы ваши переменные и программы никого не обидели, и наоборот. Пространство имен немного похоже на ваш собственный дом, в котором можно быть сколь угодно неряшливым, соблюдая, конечно, во взаимодействии с прочими гражданами достаточные приличия. В Perl пространство имен называется *пакетом* (*package*). Пакеты образуют те базовые строительные блоки, из которых сооружаются понятия модулей и классов, находящиеся на более высоком уровне.

Как и понятие «дом», понятие «пакет» несколько туманно. Пакеты не зависят от файлов. Можно иметь много пакетов в одном файле и один пакет, охватывающий несколько файлов, точно так же, как ваш дом может быть маленькой мансардой в большом здании (если вы – голодающий художник), а может состоять из нескольких зданий (если вас зовут королева Елизавета). Но обычный размер дома – одно здание, а обычный размер пакета – один файл. Perl оказывает особую помощь тем, кто хочет поместить пакет в один файл, если только они согласны дать файлу то же имя, что и пакету и использовать расширение *.pm*, что представляет собой сокращение от «perl module». *Модуль* является в Perl фундаментальной единицей повторного использования кода. На практике обращение к модулю осуществляется с помощью директивы компилятора `use`, управляющей импортом из модуля подпрограмм и переменных. Все виденные вами до сих пор случаи применения `use` были примерами повторного использования модулей.

Архив Perl (Comprehensive Perl Archive Network, или CPAN) – то место, куда вы должны поместить свои модули, если считаете, что они могут оказаться полезными другим. Процветание Perl вызвано готовностью программистов поделиться плодами своего труда с обществом. Естественно, на CPAN можно найти модули, заботливо помещенные туда другими для всеобщего пользования. Подробности см. в разделе «CPAN» главы 22 и на *www.cpan.org*.

Тенденцией последних примерно 25 лет была разработка языков программирования, навязывающих состояние паранойи. Предполагается, что каждый модуль вы должны программировать так, будто он находится в положении осады. Конечно, есть некоторые феодальные культуры, к которым это подходит, но не все культуры таковы. В культуре Perl, к примеру, предполагается, что вы не залезаете в чужой дом потому, что вас не приглашали, а не потому, что на окнах стоят решетки.[1]

Эта книга не посвящена объектно-ориентированным технологиям, и мы не собираемся обращать вас в неистового фанатика объектной ориентированности, даже если вы этого хотите. Для этого уже существует масса книг. Философия Perl в отношении объектно-ориентированной разработки соот-

[1] Однако в Perl есть некоторые решетки, если они вам нужны. См. раздел «Работа с небезопасным кодом» главы 23 «Защита данных».

ветствует его философии в отношении всего прочего: прибегать к объектно-ориентированной разработке там, где это имеет смысл, и избегать там, где этого смысла нет. Выбор за вами.

На жаргоне ООП каждый объект принадлежит объединению, называемому *классом*. В Perl классы, пакеты и модули так тесно связаны, что новичкам часто кажется, будто они взаимозаменяемы. Типичный класс реализуется модулем, в котором определен пакет, имя которого совпадает с именем класса. Все это мы разъясним в нескольких последующих главах.

Применение модулей дает преимущество непосредственного повторного использования программного обеспечения. А классы обеспечивают преимущество косвенного повторного использования программного обеспечения, когда один класс использует другой посредством наследования. Применение классов делает возможным даже большее: гладкое взаимодействие с другим пространством имен. Доступ ко всему, что находится в классе, осуществляется косвенным образом, изолируя класс от внешнего мира.

Как упоминалось в главе 8 «Ссылки», объектно-ориентированное программирование в Perl реализуется через ссылки, объекты которых «знают», к какому классу они принадлежат. Фактически, теперь, когда вы знаете о ссылках, вы знаете почти обо всем сложном, что есть в объектах. Все остальное «находится в пальцах», как сказал бы пианист. Конечно, нужно немного попрактиковаться.

Одно из основных упражнений для пальцев состоит в том, чтобы научиться защищать различные фрагменты кода от непреднамеренного взаимного воздействия на переменные. Каждый участок кода принадлежит определенному *пакету*, который определяет, какие переменные и подпрограммы ему доступны. Когда Perl встречает фрагмент кода, то компилирует его в то, что называется *текущим пакетом*. Первоначальный текущий пакет называется «`main`», но в любой момент вы можете переключить текущий пакет на другой с помощью объявления `package`. Текущий пакет определяет, какая таблица имен используется для поиска переменных, подпрограмм, дескрипторов ввода/вывода и форматов.

Любые переменные, не объявленные с ключевым словом `my`, ассоциируются с пакетом – даже кажущиеся вездесущими переменные типа `$_` и `%SIG`. На самом деле такой вещи, как глобальная переменная, в Perl фактически нет. (Специальные идентификаторы, такие как `_` и `SIG`, просто кажутся глобальными, поскольку их пакетом по умолчанию является `main`, а не текущий.)

Область видимости объявления пакета простирается от самого объявления и до конца охватывающей области (блока, файла или `eval` – что раньше встретится) или до объявления другого пакета на том же уровне, заменяющего более раннее. (Это обычный прием.)

Все последующие идентификаторы (включая объявленные `our`, но исключая объявленные `my` или квалифицированные именем другого пакета) будут помещены в таблицу имен, принадлежащую текущему пакету. (Переменные, объявленные с помощью `my`, не зависят от пакетов; они всегда видимы

внутри своей охватывающей области видимости и только в ней, независимо от объявлений пакетов.)

Обычно объявление пакета является первой командой файла, который должен включаться посредством `require` или `use`. Но, опять-таки, это лишь соглашение. Объявление пакета можно поместить в любое место, где может находиться команда. Можно даже поместить его в конце блока, и тогда оно вообще не будет иметь никакого эффекта. Можно переключать пакет в нескольких местах; объявление пакета просто выбирает таблицу имен, которая будет использоваться компилятором в оставшейся части блока. (Этот способ позволяет создать пакет, охватывающий более одного файла.)

На идентификаторы[1] в других пакетах можно ссылаться, ставя префикс идентификатора в виде имени пакета и удвоенного двоеточия: `$Пакет::Переменная` («квалифицируя» его). Если имя пакета пусто, предполагается использование пакета `main`. Это значит, что `$::sail` эквивалентно `$main::sail`.[2]

Прежним ограничителем пакетов была одинарная кавычка, поэтому в старых программах Perl можно увидеть переменные типа `$main'sail` и `$somepack'horse`. Но сейчас предпочтительным ограничителем является удвоенное двоеточие – отчасти потому, что его легче читать человеку, отчасти потому, что его легче читать макросам *emacs*. Он также позволяет программистам C++ чувствовать себя как дома – в противоположность использованию одинарной кавычки в качестве разделителя, которая должна была помочь в том же программистам Ada. Поскольку старомодный синтаксис все еще поддерживается для обратной совместимости, то в такой строке, как `"This is $owner's house"`, будет выполнено обращение к `$owner::s`; т. е. переменной `$s` в пакете `owner`, чего вы, возможно, не желали. Двусмысленности можно избежать при помощи фигурных скобок, например, `"This is ${owner}'s house"`.

Удвоенное двоеточие может применяться для связывания в цепочку идентификаторов в имени пакета: `$Red::Blue::var`. Это обозначает переменную `$var`, принадлежащую пакету `Red::Blue`. Пакет `Red::Blue` не имеет никакого отношения к возможно существующим пакетам `Red` или `Blue`. То есть, связь между `Red::Blue` и `Red` или `Blue` может что-то означать для того, кто пишет или использует программу, но ничего не значит для Perl. (Кроме того обстоятельства, что в текущей реализации таблица имен `Red::Blue` хранится в таблице имен `Red`. Но язык Perl прямо этого не использует.)

---

[1] Под идентификаторами мы подразумеваем имена, используемые как ключи таблицы имен для доступа к скалярным переменным, переменным массивов, переменных хешей, подпрограмм, дескрипторов файлов и каталогов и форматов. Синтаксически говоря, метки тоже являются идентификаторами, но они не помещаются в какую-либо таблицу имен, а непосредственно прикрепляются к командам программы. Метки не могут квалифицироваться именем пакета.

[2] Чтобы избежать возможной путаницы, в имени переменной типа `$main::sail` мы используем термин «идентификатор» для `main` и `sail`, но не для `main::sail`. Мы называем это именем переменной, потому что идентификаторы не могут содержать двоеточия.

По этой причине каждое объявление package должно объявлять полное имя пакета. Имена пакетов не допускают каких-либо подразумеваемых «префиксов», даже если (как кажется) они объявлены в области видимости объявления какого-либо другого пакета.

Только идентификаторы (имена, начинающиеся с буквы или символа подчеркивания) хранятся в таблице имен пакета. Все остальные имена хранятся в пакете main, в том числе все небуквенные переменные, такие как $!, $? и $_. Кроме того, идентификаторы STDIN, STDOUT, STDERR, ARGV, ARGVOUT, ENV, INC и SIG, если они не квалифицированы, принудительно помещаются в пакет main, даже если используются для других целей, нежели их встроенные тезки. Не называйте свои пакеты m, s, y, tr, q, qq, qr, qw или qx, если только не хотите нажить себе множество неприятностей. Например, вы не сможете использовать квалифицированную форму идентификатора в качестве дескриптора файла, поскольку она будет интерпретирована как поиск по шаблону, подстановка или транслитерация.

Давным-давно переменные, начинавшиеся с символа подчеркивания, помещались в пакет main, но мы решили, что авторам пакетов удобнее при помощи ведущего символа подчеркивания обозначать полузакрытые идентификаторы, предназначенные только для использования внутри пакета. (Подлинно закрытые переменные могут быть объявлены как лексические с областью видимости в файле, но это лучше всего работает, когда между пакетом и модулем имеется взаимно однозначное соответствие, что бывает часто, но не всегда.)

Хеш %SIG (предназначенный для перехвата сигналов; см. главу 16 «Межпроцессное взаимодействие») тоже является особым. Если определить обработчик сигнала как строку, то она будет указывать на подпрограмму в пакете main, если только явно не задано имя другого пакета. Задавайте полностью квалифицированное имя обработчика сигнала, если вы хотите указать конкретный пакет, или вообще избегайте использования строк, осуществляя вместо этого присвоение typeglob или ссылки на функцию:

```
$SIG{QUIT} = "Pkg::quit_catcher"; # полностью квалифицированное имя обработчика
$SIG{QUIT} = "quit_catcher";      # подразумевается "main::quit_catcher"
$SIG{QUIT} = *quit_catcher;       # устанавливает sub из текущего пакета
$SIG{QUIT} = \&quit_catcher;      # устанавливает sub из текущего пакета
$SIG{QUIT} = sub { print "Перехвачен SIGQUIT\n" };   # анонимная sub
```

Понятие «текущего пакета» относится как к этапу компиляции, так и к этапу исполнения. Большинство операций поиска имен переменных происходит на этапе компиляции, а поиск на этапе исполнения случается при разыменовании символических переменных, а также когда новые фрагменты кода анализируются при выполнении eval. В частности, когда eval применяется к строке, Perl знает, в каком пакете была вызвана eval, и распространяет действие этого пакета внутри обрабатываемой строки. (Конечно, внутри строки eval всегда можно переключиться в другой пакет, поскольку строка eval считается блоком, как и файл, загруженный с помощью do, require или use.)

С другой стороны, eval может узнать, в каком пакете она находится, с помощью специального символа __PACKAGE__, в котором содержится имя текущего пакета. Поскольку допускается обращение с ним как со строкой, можно использовать его в символической ссылке для доступа к переменной пакета. Но в таком случае, вероятно, следовало бы объявить переменную с ключевым словом our, чтобы к ней можно было обращаться как к лексической переменной.

Таблицы имен

Содержимое пакета в совокупности называется *таблицей имен* (*symbol table*). Таблица имен хранится в хеше, имя которого совпадает с именем пакета, но дополнено двумя двоеточиями. Таким образом, именем таблицы для main служит %main::. Поскольку main оказывается и пакетом по умолчанию, Perl допускает сокращенное имя %:: для %main::.

Аналогично, таблица имен для пакета Red::Blue носит имя %Red::Blue::. Таблица имен main содержит все остальные таблицы имен верхнего уровня, включая саму себя, поэтому %Red::Blue:: в то же время является и %main::Red::Blue::.

Когда мы говорим, что таблица имен содержит другую таблицу имен, мы имеем в виду, что она содержит ссылку на другую таблицу имен. Поскольку main является пакетом верхнего уровня, он содержит ссылку на самого себя, поэтому %main:: это то же самое, что и %main::main::, и %main::main::main::, и т. д. до бесконечности. Этот особый случай важно учитывать при написании кода, который обходит все таблицы имен.

Внутри хеша таблицы имен каждая пара ключ/значение устанавливает соответствие между именем переменной и ее значением. Ключи представляют собой символические идентификаторы, а значения являются соответствующими typeglob. Поэтому посредством обозначения *NAME typeglob мы, в действительности, просто обращаемся к значению в хеше, который содержит таблицу имен текущего пакета. На самом деле следующие строки имеют (почти) одинаковый эффект:

```
*sym = *main::variable;
*sym = $main::{"variable"};
```

Первая строка более эффективна, потому что доступ к таблице имен main может осуществляться на этапе компиляции. Она также создает новый typeglob с указанным именем, если его еще не существует, а вторая форма – нет.

Поскольку пакет представляет собой хеш, можно найти ключи пакета и получить все его переменные. А так как значения хеша являются typeglob, разыменовывать их можно несколькими способами. Например, так:

```
foreach $symname (sort keys %main::) {
    local *sym = $main::{$symname};
    print "\$$symname определен\n" if defined $sym;
```

```
    print "\@$symname не нуль\n" if         @sym;
    print "\%$symname не нуль\n" if         %sym;
}
```

Все пакеты доступны (прямо или косвенно) через пакет main, поэтому можно написать код Perl, который обойдет все переменные программы. Именно это делает отладчик Perl, когда вы запрашиваете у него дамп переменных с помощью команды V. Заметьте, что при этом не будут видны переменные, объявленные с помощью my, поскольку они не зависят от пакетов, но будут видны переменные, объявленные через our. См. главу 20 «Отладчик».

Ранее мы сказали, что только идентификаторы могут храниться в пакетах, отличных от main. Это маленький обман: в качестве ключа в хеше таблицы имен можно указать любую строку, просто Perl не допустит непосредственного использования строки, не являющейся идентификатором:

```
$!@#$%            = 0;          # НЕВЕРНО, синтаксическая ошибка.
${'!@#$%'}        = 1;          # Ok, хотя не квалифицирована.

${'main::!@#$%'} = 2;           # Можно квалифицировать в строке.
print ${ $main::{'!@#$%'} }     # Ok, выводит 2!
```

Присвоение для typeglob осуществляет операцию создания псевдонима, т. е. в результате

```
*dick = *richard;
```

переменные, подпрограммы, форматы, дескрипторы файлов и каталогов, доступные через идентификатор richard, оказываются доступными и через имя dick. Для того чтобы создать псевдоним только для конкретной переменной или подпрограммы, используйте ссылку:

```
*dick = \$richard;
```

В результате $richard и $dick становятся одной и той же переменной, но @richard и @dick остаются разными массивами. Ловко, не правда ли?

Вот как работает Exporter при импорте имен из одного пакета в другой. Например,

```
*SomePack::dick = \&OtherPack::richard;
```

импортирует функцию &richard из пакета OtherPack в пакет SomePack, делая ее доступной как функцию &dick. (Модуль Exporter описывается в следующей главе.) Если перед присвоением поставить local, то псевдоним будет действовать только до конца текущей динамической области видимости.

Этот механизм может применяться для извлечения ссылки из подпрограммы, делая объект ссылки доступным в качестве соответствующего типа данных:

```
*units = populate() ;           # Присвоить \%newhash для typeglob
print $units{kg};               # Выводит 70; разыменование не требуется!

sub populate {
```

```
    my %newhash = (km => 10, kg => 70);
    return \%newhash;
}
```

Аналогично можно передать ссылку в подпрограмму и использовать ее без разыменования:

```
%units = (miles => 6, stones => 11);
fillerup( \%units );          # Передача ссылки
print $units{quarts};         # Выводит 4
sub fillerup {
    local *hashsym = shift;   # Присвоить \%units для typeglob
    $hashsym{quarts} = 4;     # Действует на %units; не требуется разыменования!
}
```

Это ловкие приемы для передачи ссылок без лишних затрат, когда вы не хотите явно их разыменовывать. Обратите внимание на то, что оба приема работают только с переменными пакета; их нельзя использовать, если объявить `%units` как `my`.

С помощью таблиц имен можно также создавать скалярные «константы»:

```
*PI = \3.14159265358979;
```

Теперь изменить `$PI` нельзя, что, вообще-то, правильно. Это не то же самое, что подпрограмма-константа, оптимизируемая на этапе компиляции. Подпрограмма-константа – это такая подпрограмма, в прототипе которой указано, что она не принимает аргументов и возвращает постоянное выражение; подробности можно найти в разделе «Подставляемые функции-константы» главы 6 «Подпрограммы». Удобным сокращением является использование директивы `use constant` (см. главу 31 «Модули прагм»):

```
use constant PI => 3.14159;
```

При реализации используется место для подпрограммы `*PI`, а не место для скаляра, как ранее. Это эквивалентно более компактной (но хуже читаемой) записи:

```
*PI = sub () { 3.14159 };
```

Однако это удобная идиома, которую нужно знать: присвоение `sub {}` для typeglob является способом дать имя анонимной подпрограмме на этапе выполнения.

Присвоить ссылку на typeglob другой переменной typeglob (`*sym = \*oldvar`) означает то же самое, что присвоить всю typeglob, потому что Perl автоматически разыменовывает ссылку на typeglob. А когда typeglob присваивается обычная строка, она становится именем typeglob в целом, потому что Perl ищет строку в текущей таблице имен. Все нижеследующие строки эквивалентны между собой, но первые две вычисляют элемент таблицы имен на этапе компиляции, а вторые две – на этапе исполнения:

```
*sym =   *oldvar;
*sym =  \*oldvar;        # автоматическое разыменование
*sym = *{"oldvar"};      # явный поиск в таблице имен
*sym =   "oldvar";       # неявный поиск в таблице имен
```

Осуществляя любое из следующих присваиваний, мы заменяем только одну ссылку внутри typeglob:

```
*sym = \$frodo;
*sym = \@sam;
*sym = \%merry;
*sym = \&pippin;
```

Если взглянуть на это со стороны, то typeglob можно рассматривать как хеш, содержащий записи для различных типов переменных. В данном случае ключи фиксированы, поскольку typeglob может содержать ровно один скаляр, один массив, один хеш и т. д. Но можно извлекать отдельные ссылки:

```
*pkg::sym{SCALAR}      # то же, что \$pkg::sym
*pkg::sym{ARRAY}       # то же, что \@pkg::sym
*pkg::sym{HASH}        # то же, что \%pkg::sym
*pkg::sym{CODE}        # то же, что \&pkg::sym
*pkg::sym{GLOB}        # то же, что \*pkg::sym
*pkg::sym{IO}          # внутренний дескриптор файла/каталога, нет прямого
                         эквивалента
*pkg::sym{NAME}        # "sym" (не ссылка)
*pkg::sym{PACKAGE}     # "pkg" (не ссылка)
```

Выражения `*foo{PACKAGE}` и `*foo{NAME}` позволяют узнать, каковы имя и пакет для элемента таблицы имен `*foo`. Это может оказаться полезным в подпрограмме, которой typeglob передаются в качестве аргументов:

```
sub identify_typeglob {
    my $glob = shift;
    print 'Вы передали мне ', *{$glob}{PACKAGE}, '::', *{$glob}{NAME}, "\n";
}

identify_typeglob(*foo);
identify_typeglob(*bar::glarch);
```

В результате выводится:

```
Вы передали мне main::foo
Вы передали мне bar::glarch
```

Обозначение `*foo{`*THING*`}` можно использовать для получения ссылок на отдельные элементы `*foo`. Подробности см. в разделе «Ссылки на таблицы имен» главы 8 «Ссылки».

Этот синтаксис служит, в основном, для получения внутренней ссылки на дескриптор файла или каталога, поскольку другие внутренние ссылки можно получить иным способом. (Прежнее `*foo{FILEHANDLE}` все же поддерживается в значении `*foo{IO}`, только не думайте, что это имя позволит отличить дескрипторы файлов от дескрипторов каталогов.) Но мы решили, что сле-

дует обобщить его, поскольку оно выглядит довольно приятно. Скорее всего, вам не нужно все это запоминать, если только вы не собираетесь написать новый отладчик Perl.

Автозагрузка

Обычно невозможно вызвать подпрограмму, которая не определена. Однако если в пакете с неопределенной подпрограммой есть подпрограмма с именем AUTOLOAD (или если речь идет о методе объекта, и она есть в пакете какого-нибудь из базовых классов объекта), то подпрограмма AUTOLOAD вызывается с теми же аргументами, которые были бы переданы исходной подпрограмме. Можно определить подпрограмму AUTOLOAD так, чтобы она возвращала значения как обычная подпрограмма, или заставить ее определить отсутствующую подпрограмму, а затем вызвать ее, как будто она все время была на месте.

Полностью квалифицированное имя исходной подпрограммы чудесным образом появляется в $AUTOLOAD – глобальной переменной того же пакета, в котором находится подпрограмма AUTOLOAD. Вот простой пример, который деликатно предупреждает вас о вызове неопределенной подпрограммы, вместо того чтобы осуществить выход:

```
sub AUTOLOAD {
    our $AUTOLOAD;
    warn "Попытка вызвать $AUTOLOAD не удалась.\n";
}

blarg(10);              # наша $AUTOLOAD получит значение main::blarg
print "Все еще живы!\n";
```

Либо можно возвратить значение от имени не определенной подпрограммы:

```
sub AUTOLOAD {
    our $AUTOLOAD;
    return "Я вижу $AUTOLOAD(@_)\n";
}

print blarg(20);        # выводит: Я вижу main::blarg(20)
```

Ваша подпрограмма AUTOLOAD может загрузить определение для неопределенной подпрограммы с помощью eval или require, либо прибегнуть к обсуждавшемуся выше приему с присвоением glob, а затем выполнить эту неопределенную подпрограмму с применением специальной формы goto, которая может без следа стереть запись активации подпрограммы AUTOLOAD. Сейчас мы определим подпрограмму, присвоив замыкание переменной glob:

```
sub AUTOLOAD {
    my $name = our $AUTOLOAD;
    *$AUTOLOAD = sub { print "Я вижу $name(@_)\n" };
    goto &$AUTOLOAD;    # Запустить новую подпрограмму.
}

blarg(30);              # выводит: Я вижу main::blarg(30)
```

```
glarb(40);              # выводит: Я вижу main::glarb(40)
blarg(50);              # выводит: Я вижу main::blarg(50)
```

Стандартный модуль AutoSplit применяется разработчиками модулей для автоматического разделения своих модулей на отдельные файлы (имена которых заканчиваются *.al*), в каждом из которых содержится одна подпрограмма. Эти файлы помещаются в каталог *auto/* в системной библиотеке Perl, после чего их можно автоматически загружать по требованию с помощью стандартного модуля AutoLoader.

Аналогичный подход осуществлен в модуле SelfLoader, за исключением того, что он автоматически загружает функции из собственной области DATA файла, что менее эффективно в одних отношениях и более эффективно в других. Автозагрузка функций Perl с помощью AutoLoader и SelfLoader аналогична динамической загрузке скомпилированных функций C с помощью DynaLoader, за исключением того, что минимальная единица автозагрузки – функция, а минимальная единица динамической загрузки – целый модуль, и при этом обычно компонуется сразу много функций C или C++. (Многие программисты Perl прекрасно обходятся без модулей AutoSplit, AutoLoader, SelfLoader или DynaLoader. Просто знайте, что они есть, на случай, если *не сможете* прекрасно обходиться без них.)

Можно неплохо развлечься, работая с программами AUTOLOAD, служащими оболочками для других интерфейсов. Давайте, например, потребуем, чтобы для каждой не определенной функции вызывалась команда system с теми же аргументами. Нужно сделать лишь следующее:

```
sub AUTOLOAD {
    my $program = our $AUTOLOAD;
    $program =~ s/.*::/;  # обрезать имя пакета
    system($program, @_);
}
```

(Поздравляем, вы реализовали элементарную форму модуля Shell, стандартно поставляемого с Perl.) Вызвать свой автозагрузчик (в Unix) можно так:

```
date();
who('am', 'i');
ls('-l');
echo("Abadugabudabuda...");
```

На самом деле, если предварительно объявить функции, которые требуется вызывать таким способом, можно действовать так, как если бы они были встроенными, и опускать скобки при вызове:

```
sub date (;$$);        # Допускает от нуля до двух аргументов.
sub who (;$$$$);       # Допускает от нуля до четырех аргументов.
sub ls;                # Допускает любое число аргументов.
sub echo ($@);         # Допускает не меньше одного аргумента.

date;
who "am", "i";
ls "-l";
echo "That's all, folks!";
```

11

Модули

Модуль является в Perl фундаментальной единицей повторно используемого кода. Он реализуется просто как пакет, определенный в файле с тем же именем (и *.pm* в конце). В этой главе мы обсудим, как можно использовать созданные другими модули и создавать собственные.

Perl поставляется с большим числом модулей, которые можно найти в каталоге *lib* дистрибутива Perl. Многие из этих модулей описаны в главе 32 «Стандартные модули» и главе 31 «Модули прагм». Для всех стандартных модулей имеется также обширная электронная документация, которая (о, ужас!) может быть более актуальной, чем данная книга. Попробуйте выполнить команду *perldoc*, если команда *man* не работает.

Архив Perl «Comprehensive Perl Archive Network» (CPAN) содержит общемировое хранилище модулей, созданных сообществом Perl, и обсуждается в главе 22 «CPAN». См. также *http://www.cpan.org*.

Использование модулей

Есть два вида модулей: традиционные и объектно-ориентированные. Традиционные модули определяют подпрограммы и переменные, которые вызывающая их сторона может импортировать и затем использовать. Объектно-ориентированные модули действуют как определения классов, и доступ к ним осуществляется через вызов методов, как описывается в главе 12 «Объекты». Некоторые модули делают и то и другое.

Модули Perl обычно включаются в программу с помощью команды:

```
use MODULE LIST;
```

или просто:

```
use MODULE;
```

MODULE должен быть идентификатором, соответствующим имени пакета модуля и файла. (Описания синтаксиса здесь предполагаются наводящими; полный синтаксис команды use дан в главе 29 «Функции».)

Команда use осуществляет предварительную загрузку модуля *MODULE* на этапе компиляции и импорт запрошенных имен, которые становятся доступными на протяжении последующей компиляции. Если не задать список *LIST* нужных имен, используются имена, которые содержатся во внутреннем массиве модуля @EXPORT – в предположении, что применяется модуль Exporter, описываемый в разделе «Закрытость модуля и Exporter» далее в этой главе. (Если *LIST* указан, то все его имена должны содержаться в массивах @EXPORT или @EXPORT_OK, иначе возникнет ошибка.)

Если модули служат для импорта имен в текущий пакет Exporter, можно использовать имена модуля без квалификатора пакета:

```
use Fred;        # Если в Fred.pm есть @EXPORT = qw(flintstone),
flintstone();    # ...это вызывает Fred::flintstone().
```

Все модули Perl имеют расширение *.pm*. Это (как и кавычки) предполагается обеими командами use и require, поэтому вам не нужно писать "*MODULE*.pm". Применение голых идентификаторов помогает отличить новые модули от библиотек *.pl* и *.ph*, применявшихся в старых версиях Perl. При этом *MODULE* вводится также в качестве официального имени модуля, что помогает анализатору в некоторых двусмысленных ситуациях. Удвоенное двоеточие в имени модуля транслируется в разделитель каталогов, используемый системой, поэтому если ваш модуль называется Red::Blue::Green, то Perl будет искать его как *Red/Blue/Green.pm*.

Perl ищет модули в каждом из каталогов, перечисленных в массиве @INC. Поскольку use загружает модули на этапе компиляции, любые изменения @INC должны происходить тоже на этапе компиляции. Сделать это можно с помощью директивы lib, описываемой в главе 32 «Модули прагм» или с помощью блока BEGIN. После включения модуля в хеш %INC добавляется пара ключ/значение. Ключом будет имя файла модуля (в нашем примере Red/Blue/Green.pm), а значением – полный путь, которым может быть что-то вроде C:/perl/site/lib/Red/Blue/Green.pm для модуля, надлежащим образом установленного под Windows.

Имена модулей должны начинаться с заглавной буквы, если только они не действуют в качестве директивных. Модули прагм действуют как директивы компилятора (советы компилятору), поэтому мы резервируем имена

директив (директивных модулей) на нижнем регистре для использования в дальнейшем.

При загрузке модуля посредством use любой код в нем выполняется так же, как при загрузке с помощью обычного require. Если вам безразлично, когда загружается модуль – при компиляции или на этапе исполнения, можете просто сказать:

```
require MODULE;
```

В целом, однако, предпочтительнее использовать use, а не require, поскольку в результате поиск модулей происходит при компиляции и, если есть ошибки, о них можно узнать раньше.

Следующие две команды выполняют почти одно и то же:

```
require MODULE;
require "MODULE.pm";
```

Однако есть два отличия. В первой команде require транслирует удвоенные двоеточия в системный разделитель каталогов – так же, как это делает use. Во втором случае трансляция не выполняется, что вынуждает указывать имя пути к модулю литерально и делает код менее переносимым. Второе отличие в том, что первая команда require сообщает компилятору, что выражения с косвенными обозначениями объектов, включающие в себя *MODULE* (например, $ob = purge *MODULE*), являются вызовами методов, а не функций. (Это действительно может быть существенно, если в вашем собственном модуле имеется конфликтующее определение purge.)

Поскольку объявление use и родственное объявление no подразумевают наличие блока BEGIN, компилятор загружает модуль (и выполняет имеющийся в нем код инициализации), как только обнаруживает это объявление и прежде чем компилировать оставшуюся часть файла. Благодаря этому директивы могут влиять на работу компилятора, а модули объявлять подпрограммы, которые затем видны как списковые операторы на протяжении дальнейшей компиляции. Этого не будет, если вместо use использовать require. Едва ли не единственной причиной для применения require является наличие двух модулей, в каждом из которых требуется функция из другого модуля. (И мы не уверены, что это достаточное основание.)

Модули Perl всегда загружают файл *.pm*, но этот файл, в свою очередь, может загружать связанные файлы, например, динамически компонуемые библиотеки C или C++ или автоматически загружаемые определения подпрограмм Perl. При этом дополнительные файлы будут полностью прозрачны для пользователя модуля. Обязанность загрузки (или подготовки к автоматической загрузке) дополнительных функций лежит на файле *.pm*. Модуль POSIX осуществляет как динамическую, так и автоматическую загрузку, но можно просто сказать:

```
use POSIX;
```

и получить все экспортируемые функции и переменные.

Создание модулей

Ранее мы сказали, что у модуля есть два способа сделать свой интерфейс доступным вашей программе: экспортируя имена или разрешая вызовы методов. Здесь мы приведем пример первого стиля; второй стиль предназначен для объектно-ориентированных модулей и описывается в следующей главе. (Объектно-ориентированные модули не должны ничего экспортировать, т. к. вся идея методов заключается в том, что Perl находит их автоматически, исходя из типа объекта.)

Для конструирования модуля с именем `Bestiary`, создайте файл *Bestiary.pm*, выглядящий так:

```
package      Bestiary;
require      Exporter;

our @ISA       = qw(Exporter);
our @EXPORT    = qw(camel);     # Имена, экспортируемые по умолчанию
our @EXPORT_OK = qw($weight);   # Имена, экспортируемые по запросу
our $VERSION   = 1.00;          # Номер версии

### Здесь включаются ваши функции и переменные

sub camel { print "Одногорбый верблюд" }

$weight = 1024;

1;
```

Теперь программа может сказать `use Bestiary`, чтобы получить доступ к функции `camel` (но не к переменной `$weight`), и сказать `use Bestiary qw(camel $weight)`, чтобы получить доступ как к функции, так и к переменной.

Можно также создавать модули, динамически загружающие код, написанный на C. Подробности см. в главе 21 «Внутри и снаружи».

Закрытость модуля и Exporter

Perl не осуществляет автоматического патрулирования границ между закрытыми и открытыми элементами в своих модулях – в отличие от таких языков, как C++, Java и Ada, Perl не озабочен поддержкой закрытости. Модуль Perl предпочтет, чтобы вы не входили в его комнату, потому что вас не приглашали, а не потому, что у него есть дробовик.

Между модулем и его пользователем заключен контракт, часть которого составляет общее право, а часть записывается. Общее право устанавливает, что модуль будет воздерживаться от изменения любого пространства имен, если его об этом не просили. Письменная часть контракта для модуля (а именно документация) может ставить дополнительные условия. И тогда после прочтения контракта предполагается, что, говоря `RedefineTheWorld`, вы знаете, что изменяете мир и готовы к риску, связанному с последствиями этого. Самый распространенный способ переопределения миров – это ис-

пользование модуля Exporter. Как вы увидите далее в этой главе, с помощью данного модуля можно даже переопределять встроенные функции.

Если модуль загружается с помощью use, то обычно он делает некоторые переменные и функции доступными вашей программе, или, точнее, ее текущему пакету. Это действие по экспортированию имен модуля (и, таким образом, импортированию их в вашу программу) иногда называют *загрязнением* (*polluting*) вашего пространства имен. Большинство модулей использует для этого Exporter; поэтому в начале большинства модулей есть что-нибудь вроде:

```
require Exporter;
our @ISA = ("Exporter");
```

Эти две строки заставляют модуль наследовать классу Exporter. Наследование описывается в следующей главе, но вам нужно только знать, что наш модуль Bestiary может экспортировать имена в другие пакеты с помощью таких строк:

```
our @EXPORT    = qw($camel %wolf ram);              # Экспорт по умолчанию
our @EXPORT_OK = qw(leopard @llama $emu);           # Экспорт по запросу
our %EXPORT_TAGS = (                                # Экспорт как группы
                     camelids => [qw($camel @llama)],
                     critters => [qw(ram $camel %wolf)],
                   );
```

С точки зрения экспортирующего модуля массив @EXPORT содержит имена переменных и функций, экспортируемых по умолчанию: то, что ваша программа получает, сказав use Bestiary. Переменные и функции в @EXPORT_OK экспортируются только тогда, когда программа специально запрашивает их в команде use. Наконец, пары ключ/значение в %EXPORT_TAGS позволяют программе включать отдельные группы имен, перечисленных в @EXPORT и @EXPORT_OK.

С точки зрения импортирующего пакета, команда use задает список имен, которые нужно импортировать, группу, указанную в %EXPORT_TAGS, шаблон имен или вообще ничего, и тогда в программу импортируются имена, указанные в @EXPORT.

Для импорта имен из модуля Bestiary можно включить любую из следующих команд:

```
use Bestiary;                    # Импортировать имена из @EXPORT
use Bestiary ();                 # Ничего не импортировать
use Bestiary qw(ram @llama);     # импортировать функцию ram и массив @llama
use Bestiary qw(:camelids);      # Импортировать $camel и @llama
use Bestiary qw(:DEFAULT);       # Импортировать имена @EXPORT
use Bestiary qw(/am/);           # Импортировать $camel, @llama и ram
use Bestiary qw(/^\$/);          # Импортировать все скаляры
use Bestiary qw(:critters !ram); # Импортировать critters, за исключением ram
use Bestiary qw(:critters !:camelids);
                                 # Импортировать critters, за исключением camelids
```

Отсутствие элемента в списке экспорта (или явное удаление из списка импорта с помощью восклицательного знака) не делает его недоступным программе, использующей модуль. Программа всегда может обратиться к содержимому пакета модуля, полностью квалифицировав его именем пакета, например, `%Bestiary::gecko`. (Поскольку лексические переменные не принадлежат пакетам, возможность закрытия сохраняется: см. раздел «Закрытые методы» главы 12.)

Чтобы увидеть, как обрабатываются спецификации и что действительно импортируется в ваш пакет, можно сказать `BEGIN { $Exporter::Verbose=1 }`.

`Exporter` сам представляет собой модуль Perl, и, если у вас есть интерес, вы можете посмотреть, какие фокусы он выделывает с typeglob для экспорта имен из одного пакета в другой. Внутри модуля `Exporter` ключевой является функция `import`, которая осуществляет назначение псевдонимов, необходимое для того, чтобы имя одного пакета появилось в другом. На практике команда `use Bestiary` *`LIST`* совершенно эквивалентна

```
BEGIN {
    require Bestiary;
    import Bestiary LIST;
}
```

Это значит, что вашим модулям не обязательно обращаться к `Exporter`. Модуль может делать все то же, что при его использовании, т. к. `use` просто вызывает обычный метод `import` для модуля, а вы можете определить этот метод так, чтобы он выполнял все что вам угодно.

Экспортирование без вызова метода import модуля Exporter

`Exporter` определяет метод с именем `export_to_level`, применяемый в ситуациях, когда (по какой-то причине) вы не можете непосредственно вызвать метод `import` модуля `Exporter`. Метод `export_to_level` вызывается так:

```
MODULE->export_to_level($where_to_export, @what_to_export);
```

где `$where_to_export` представляет собой целое число, указывающее, насколько далеко в стек вызовов должны экспортироваться ваши имена, а `@what_to_export` — это массив, в котором перечислены экспортируемые имена (обычно `@_`).

Предположим, например, что в нашем `Bestiary` есть своя функция `import`:

```
package Bestiary;
@ISA = qw(Exporter);
@EXPORT_OK = qw ($zoo);

sub import {
    $Bestiary::zoo = "зверинец";
}
```

Наличие этой функции import не позволяет наследовать функцию import из Exporter. Для того чтобы функция import из Bestiary, после того как она установит $Bestiary::zoo, вела себя точно так же, как функция import из Exporter, ее следует определить так:

```
sub import {
    $Bestiary::zoo = "зверинец";
    Bestiary->export_to_level(1, @_);
}
```

При этом имена экспортируются на один уровень «выше» текущего пакета, т. е. в любую программу или модуль, которые используют Bestiary.

Проверка версий

Если в модуле определена переменная $VERSION, то программа, использующая этот модуль, может проверить, достаточно ли свежей является его версия. Например:

```
use Bestiary 3.14;   # Bestiary должен иметь версию 3.14 или выше
use Bestiary v1.0.4; # Bestiary должен иметь версию 1.0.4 или выше
```

Эти команды преобразуются в вызовы Bestiary->require_version, которые модуль затем наследует.

Действия при обнаружении неизвестных имен

В некоторых ситуациях может потребоваться *воспрепятствовать* экспорту некоторых имен. Обычно это относится к модулям, содержащим функции или константы, не имеющие смысла в некоторых системах. Можно помешать Exporter экспортировать эти имена, поместив их в массив @EXPORT_FAIL.

Когда программа пытается импортировать одно из этих имен, Exporter дает модулю возможность обработать ситуацию, прежде чем сгенерировать ошибку. Он делает это, передавая список ошибочных имен вызываемому методу export_fail, который можно определить так (в предположении, что ваш модуль использует модуль Carp):

```
sub export_fail {
    my $class = shift;
    carp "К сожалению, эти имена недоступны: @_";
    return @_;
}
```

Exporter предоставляет метод export_fail, вызываемый по умолчанию, который просто возвращает в неизменном виде список и делает вызов use неуспешным, возбуждая исключительную ситуацию для каждого имени. Если export_fail возвращает пустой список, значит, ошибок не отмечено, и все запрошенные имена экспортированы.

Вспомогательные функции для обработки групп имен

Поскольку имена, перечисленные в %EXPORT_TAGS, должны также появиться в @EXPORT или @EXPORT_OK, в Exporter есть две функции, позволяющие добавить эти помеченные группы имен:

```
%EXPORT_TAGS = (foo => [qw(aa bb cc)], bar => [qw(aa cc dd)]);

Exporter::export_tags('foo');      # добавить aa, bb и cc в @EXPORT
Exporter::export_ok_tags('bar');   # добавить aa, cc и dd в @EXPORT_OK
```

Задание имен, не являющихся тегами, вызывает ошибку.

Замещение встроенных функций

Многие встроенные функции могут быть *замещены* (*overridden*), хотя (так же, как пробивать дыры в стенах своего дома) это следует делать только изредка и с достаточными на то основаниями. Обычно это может осуществляться в пакете, пытающемся эмулировать отсутствующие встроенные функции в системе, отличной от Unix. (Не путайте замещение с *перегрузкой* (*overloading*), которая придает встроенным операторам дополнительные объектно-ориентированные значения, но ничего не замещает. Дополнительно читайте об этом в описании модуля overload в главе 13 «Перегрузка».)

Замещение может осуществляться только путем импортирования имени из модуля – обычного предварительного объявления для этого недостаточно. Если быть совсем откровенным, то это присвоение typeglob, который запускает замещение, ссылки на код, как в *open = \&myopen. Кроме того, присвоение должно быть осуществлено в каком-нибудь другом пакете; это сделано с целью затруднить случайное замещение при создании псевдонима typeglob. Однако если вам действительно нужно сделать собственное замещение, не отчаивайтесь, потому что директива subs позволяет предварительно объявлять подпрограммы посредством синтаксиса импорта, и этими именами потом замещаются встроенные:

```
use subs qw(chdir chroot chmod chown);
chdir $somewhere;
sub chdir { ... }
```

Вообще говоря, модули не должны экспортировать такие встроенные имена, как open или chdir, в составе своего списка по умолчанию @EXPORT, т. к. эти имена могут попасть в еще чье-то пространство имен и неожиданным образом изменить семантику. Если вместо этого включить такое имя в список @EXPORT_OK, то импортеры будут вынуждены явно запрашивать замещение встроенного имени, принуждая таким образом всех к честности.

Исходные версии замещенных функций всегда доступны через псевдопакет CORE. Поэтому CORE::chdir всегда будет той версией, которая была изначально компилирована в Perl, даже если ключевое слово chdir было замещено.

Ну, почти всегда. Описанный механизм замещения совершенно умышленно ограничен тем пакетом, который запрашивает импорт. Но есть механизм с более широким спектром действия, который можно использовать при намерении заместить встроенную функцию везде, не взирая на границы пространств имен. Это достигается путем определения функции в псевдопакете `CORE::GLOBAL`. Ниже приведен пример, в котором оператор `glob` заменяется на нечто, понимающее регулярные выражения. (Учтите, что этот пример реализует не все, что нужно для чистого замещения встроенного в Perl `glob`, который ведет себя по-разному в зависимости от того, находится ли он в скалярном или списковом контексте. На практике многие встроенные функции Perl имеют такие зависящие от контекста режимы, и каждое правильно написанное замещение должно их адекватным образом поддерживать. Полностью функциональный пример замещения glob можно получить, изучая модуль `File::Glob`, поставляемый с Perl.) Тем не менее, вот асоциальная версия:

```
*CORE::GLOBAL::glob = sub {
    my $pat = shift;
    my @got;
    local *D;
    if (opendir D, '.') {
        @got = grep /$pat/, readdir D;
        closedir D;
    }
    return @got;
}

package Whatever;

print <^[a-z_]+\.pm\$>;       # показать все директивы в текущем каталоге
```

При глобальном замещении `glob` происходит вытесняющее навязывание нового (и подрывного) режима действия оператора `glob` в *каждом* пространстве имен без уведомления и одобрения модулей, являющихся владельцами этих пространств имен. Конечно, это нужно делать с крайней осторожностью – если делать вообще. Скорее всего, не делать.

Наша философия замещения такова: приятно быть важным, но важнее быть приятным.

12

Объекты

Прежде всего, необходимо разбираться в пакетах и модулях; см. главу 10 «Пакеты» и главу 11 «Модули». Требуется также понимание ссылок и структур данных; см. главу 8 «Ссылки» и главу 9 «Структуры данных». Кроме того, полезно немного знать об объектно-ориентированном программировании (ООП), поэтому в следующем разделе мы прочтем вам небольшой курс ООЖ (объектно-ориентированного жаргона).

Краткая памятка по объектно-ориентированному жаргону

Объект – это структура данных с некоторым набором режимов поведения (behaviors). Обычно мы говорим о поведении объекта, как если бы оно осуществлялось им непосредственно, иногда доходя при этом до антропоморфизма. Например, можно услышать, что прямоугольник «умеет» выводить себя на экран, или что он «знает», как рассчитать свою площадь.

Каждый объект получает свои режимы поведения благодаря тому, что является *экземпляром* (*instance*) класса. Класс определяет *методы*: режимы поведения, применимые к классу и его экземплярам. Тогда, когда это различие существенно, мы называем методы, применяемые только к конкретному объекту, *методами объекта*, а те, которые применяются к классу в целом, *методами класса*. Но это только соглашение: для Perl метод всегда только метод, отличаемый лишь по типу своего первого аргумента.

Можно рассматривать метод экземпляра как некоторое действие, осуществляемое конкретным объектом, например, вывод самого себя, копирование себя или изменение одного или нескольких своих свойств («установить название этого меча равным Андрил»). Методы класса могут осуществлять операции над совокупностью многих объектов («вывести названия всех мечей») или выполнять другие действия, не зависящие от конкретных объектов («с данного момента, когда выковывается новый меч, регистрировать его владельца в этой базе данных»). Методы, которые генерируют экземпляры (объекты) класса, называются *конструкторами* («создать меч с рукояткой, украшенной драгоценными камнями и таинственной надписью»). Обычно это методы класса («сделать мне новый меч»), но могут быть и методы объекта («сделать точную копию этого меча»).

Класс может *наследовать* (*inherit*) методы *родительских классов* (*parent classes*), называемых также *базовыми классами* (*base classes*) или *надклассами* (*superclasses*). В этом случае класс называется *производным классом* (*derived class*) или *подклассом* (*subclass*). (Еще более запутывает вопрос то, что под «базовым классом» иногда в литературе имеется в виду «самый верхний» надкласс. Мы в таком значении его не используем.) Наследование создает новый класс, который ведет себя так же, как существующий, но допускает наличие измененного или дополнительного поведения, которого нет у его родителей. При вызове метода, определения которого нет в классе, Perl автоматически ищет определение в родительских классах. Например, класс меча может наследовать свой метод `attack` у общего класса клинка. У родителей могут быть свои родители, и Perl, при необходимости, будет вести поиск и в тех классах. Класс клинка, в свою очередь, может наследовать метод `attack` от еще более общего класса оружия.

Когда для объекта вызывается метод `attack`, итоговое поведение может зависеть от того, является ли объект мечом или стрелой. Возможно, никакого различия вообще не будет, например, если и меч, и стрела унаследовали свой режим нападения от более общего класса «оружие». Но если различие в поведении есть, то механизм диспетчеризации методов выбирает метод `attack`, подходящий для рассматриваемого объекта. Полезное свойство, позволяющее всегда выбрать наиболее подходящее поведение для конкретного типа объекта, называется *полиморфизмом*. Это важный вид безразличия.

Необходимо позаботиться о внутреннем содержании своих объектов при реализации своего класса, но при *использовании* класса следует представлять себе его объекты как черные ящики. Невозможно увидеть, что находится внутри, не надо знать, как он работает, и вы взаимодействуете с ящиком лишь на его условиях: через методы, предоставляемые классом. Даже если известно, *что* эти методы делают с объектом, надо устоять перед побуждением покопаться в нем самому. Это как пульт управления телевизором: даже если вы знаете, что происходит внутри него, следует избегать праздного вмешательства в его внутреннее устройство без особых причин.

Perl позволяет вам при необходимости заглянуть внутрь объекта, находясь вне класса. Но при этом нарушается его *инкапсуляция* – принцип, согласно

которому любой доступ к объекту должен осуществляться только через его методы. Инкапсуляция отделяет опубликованный интерфейс (описывающий, как объект должен использоваться) от реализации (определяющей, как он в действительности работает). В Perl нет явной возможности интерфейса помимо этого неписаного контракта между разработчиком и пользователем. От обеих сторон ожидается проявление здравого смысла и правил приличия: пользователь полагается на документированный интерфейс, разработчик обещает его не изменять.

Perl не требует, чтобы вы придерживались какого-либо определенного стиля программирования, и не одержим идеей закрытости, как некоторые другие объектно-ориентированные языки программирования. Однако Perl одержим идеей свободы, и одна из свобод, которые Perl предоставляет программисту, это право избрать такую меру закрытости, какую он пожелает. На самом деле, Perl может обеспечить более сильную закрытость в своих классах и объектах, чем C++. Это означает, что Perl ни в чем не ограничивает программиста и, в частности, не ограничивает его право ограничивать самого себя (если вас занимают такого рода вещи). Разделы «Закрытые методы» и «Использование замыканий в закрытых объектах» далее в этой главе показывают, как можно ужесточить самодисциплину.

Конечно, объекты включают в себя значительно больше, чем о них было сейчас сказано, и об объектно-ориентированном программировании можно узнать значительно больше. Но это не входит в наши задачи. Поэтому продолжим.

Система объектов Perl

В Perl отсутствует особый синтаксис для определения объектов, классов и методов. Для реализации этих трех понятий применяются существующие конструкции.[1]

Вот несколько простых определений, которые могут показаться вам ободряющими:

Объект является просто ссылкой… э-э, объектом ссылки.

> Поскольку ссылки позволяют отдельным скалярам представлять большие совокупности данных, не должно вызывать удивления, что для всех объектов используются ссылки. Технически объект не является собственно ссылкой – это, на самом деле, то, на что указывает ссылка. Однако программисты на Perl часто не замечают этого отличия, и, поскольку мы считаем это прекрасной метонимией, то продолжим здесь использовать ее, когда это для нас удобно.[2]

---

1 *Вот он*, пример повторного использования кода!

2 Мы предпочитаем лингвистическую живость математической строгости. Можете не соглашаться.

Класс является просто пакетом.

Пакет служит классом благодаря использованию его подпрограмм для выполнения методов класса и использованию переменных пакета для хранения глобальных данных класса. Часто один или несколько классов хранят в модуле.

Метод является просто подпрограммой.

Вы просто объявляете подпрограммы в пакете, используемом в качестве класса, и они будут использоваться в качестве методов класса. Вызов метода – новый способ обращения к подпрограммам – передает дополнительный аргумент: объект или пакет, используемый для вызова метода.

Вызов методов

Если бы потребовалось выделить квинтэссенцию объектно-ориентированного программирования, то ею стало бы понятие *абстракции* – одна общая тема, проходящая через все эти красивые слова, которые энтузиасты ОО любят обсуждать, типа полиморфизма, наследования и инкапсуляции. Мы верим в эти причудливые слова, но будем обращаться к ним с практической позиции их значения для вызова методов. Методы находятся в сердцевине системы объектов, поскольку представляют слой абстрагирования, необходимый для реализации всех этих причудливых терминов. Вместо того чтобы непосредственно обращаться к порции данных, находящейся в объекте, вызывают метод экземпляра. Вместо того чтобы непосредственно обращаться к подпрограмме в каком-то пакете, вызывают метод класса. Располагая этот уровень косвенности между использованием класса и реализацией класса, разработчик программы получает возможность свободно кромсать его внутренний механизм, не создавая особого риска нарушить работу программ, которые его используют.

Perl поддерживает две разные синтаксические формы для вызова методов. В одной используется знакомый вам стиль, всюду встречающийся в Perl, а вторая может показаться вам знакомой по другим языкам программирования. Какая бы форма вызова метода ни использовалась, подпрограмме, образующей метод, всегда передается дополнительный начальный аргумент. Если для вызова метода используется класс, этот аргумент будет именем класса. А если объект, то этот аргумент будет ссылкой на объект. Чем бы он ни был, мы будем называть его *инвокантом* (*invocant*)[1] метода. Для метода класса инвокантом является имя пакета. Для метода экземпляра инвокантом является ссылка, задающая объект.

Иными словами, инвокант – это то, что вызывает метод. Иногда в литературе по ООП его называют *агентом* (*agent*) или *актором* (*actor*). Грамматически инвокант не является ни субъектом действия, ни получателем этого действия. Он более похож на косвенное дополнение, бенефициара, в чью пользу

[1] От лат. in – в, внутрь и vocare – звать, призывать. – *Примеч. ред.*

осуществляется действие, – как слово «мне» в команде «выкуй мне меч!». Семантически можно представлять себе инвоканта либо как то, что создает вызов, либо как то, что служит предметом вызова, – в зависимости от того, что больше подходит вашему складу ума. Мы не собираемся заставлять вас думать каким-то определенным способом. (Во всяком случае не об этом.)

По большей части методы вызываются явно, но могут вызываться и неявно, когда их запускают деструкторы объектов, перегруженные операторы или связанные переменные. Если говорить точно, то это не обычные вызовы подпрограмм, а вызовы методов, автоматически запускаемые Perl от имени объекта. Деструкторы описываются далее в этой главе, перегрузка – в главе 13 «Перегрузка», а связи – в главе 14 «Связанные переменные».

Одно из различий между методами и подпрограммами состоит в моменте, когда происходит разрешение их пакетов, т. е. насколько рано (или поздно) Perl решает, какой код должен выполняться для метода или подпрограммы. Пакет подпрограммы разрешается во время компиляции, прежде чем программа начнет выполняться.[1] Напротив, разрешение пакета метода не осуществляется, пока метод не вызван фактически. (Проверка прототипов происходит на этапе компиляции, поэтому обычные подпрограммы могут использовать прототипы, а методы – нет.)

Причина, по которой пакет метода не может быть разрешен раньше, относительно проста: пакет определяется классом инвоканта, а инвокант неизвестен, пока метод не будет фактически вызван. В центре ООП лежит простая логическая цепочка: когда известен инвокант, известен класс инвоканта, а когда известен класс, известно, кому он наследует, а когда известно, кому наследует класс, известна фактическая подпрограмма, которую нужно вызвать.

За использование алгоритмов абстракции приходится платить. Из-за позднего разрешения методов объектно-ориентированное решение в Perl, скорее всего, будет выполняться медленнее, чем соответствующее решение без применения ООП. Для некоторых изощренных приемов, описываемых ниже, это может быть *значительно* медленнее. Однако многие распространенные задачи решаются путем не быстрой, а умной работы. Вот тут и начинает блистать ООП.

Вызов методов с помощью оператора стрелки

Как уже упоминалось, есть два стиля вызова методов. Первый выглядит так:

```
INVOCANT->METHOD (LIST)
INVOCANT->METHOD
```

[1] Более точно, вызов подпрограммы разрешается в некоторый typeglob, ссылка на который компилируется в дерево кодов операций. О значении этого typeglob можно договориться даже на этапе исполнения – именно это позволяет AUTOLOAD автоматически загружать подпрограммы. Обычно, однако, значение этого typeglob тоже разрешается на этапе компиляции с помощью определения подпрограммы с надлежащим именем.

По очевидной причине этот метод обычно называют стрелочной формой вызова. (Не путайте -> и =>, «двуствольную» стрелку, используемую вместо запятой.) Если есть аргументы, необходимы круглые скобки. При выполнении вызов сначала находит подпрограмму, которую совместно определяют класс *INVOCANT* и имя *METHOD*, а затем вызывает эту подпрограмму, передавая ей *INVOCANT* в качестве первого аргумента.

Если *INVOCANT* является ссылкой, то мы говорим, что *METHOD* вызывается как метод экземпляра (объекта), а если *INVOCANT* является именем пакета, то говорим, что *METHOD* вызывается как метод класса. На практике различия между тем и другим нет, за исключением того, что имя пакета более очевидным образом связано с самим классом, чем с объектами этого класса. Поверьте нам на слово, что объекты тоже знают, к какому классу они принадлежат. Очень скоро мы расскажем, как связать объект с именем класса, но использовать объекты можно и не зная этого.

Например, чтобы создать объект с помощью метода класса summon и вызвать затем с полученным объектом метод экземпляра speak, можно сказать:

```
$mage = Wizard->summon("Gandalf");  # метод класса
$mage->speak("friend");             # метод экземпляра
```

Методы summon и speak определены классом Wizard — или одним из классов, которым он наследует. Но вас это не должно беспокоить. Не вмешивайтесь в дела Волшебников.

Поскольку оператор стрелки ассоциативен слева (см. главу 3 «Унарные и бинарные операторы»), можно даже объединить две команды в одну:

```
Wizard->summon("Gandalf")->speak("friend");
```

Иногда требуется вызвать метод, не зная заранее его имени. Можно выбрать стрелочную форму вызова метода и заменить имя метода простой скалярной переменной:

```
$method = "summon";
$mage = Wizard->$method("Gandalf");  # Вызвать Wizard->summon

$travel = $companion eq "Shadowfax" ? "ide" : "walk";
$mage->$travel("seven leagues");     # Вызвать $mage->ride или $mage->walk
```

Хотя имя метода служит для его косвенного вызова, такое использование не запрещается use strict 'refs', поскольку в действительности для *всех* обращений к методам их поиск осуществляется по таблицам имен в момент разрешения.

В нашем примере мы записали в $travel имя метода, но это могла быть и ссылка на подпрограмму. При этом происходит обход алгоритма поиска метода, но иногда это как раз то что требуется. Посмотрите раздел «Закрытые методы» и обсуждение метода can в разделе «UNIVERSAL: первичный класс предков». О том, как создать ссылку на конкретный метод конкретного экземпляра, см. раздел «Замыкания» главы 8 «Ссылки».

Вызов методов с использованием косвенных объектов

Второй стиль вызова метода выглядит так:

```
METHOD INVOCANT (LIST)
METHOD INVOCANT LIST
METHOD INVOCANT
```

Круглые скобки вокруг *LIST* не обязательны; если они опущены, метод действует как списковый оператор. Поэтому можно писать такие команды с использованием данного стиля вызова методов:

```
$mage = summon Wizard "Gandalf";
$nemesis = summon Balrog home => "Moria", weapon => "whip";

move $nemesis "bridge";
speak $mage "You cannot pass";
break $staff;               # надежнее использовать: break $staff ();
```

Синтаксис спискового оператора должен быть вам знаком; это тот же стиль, который используется для передачи дескрипторов файлов в `print` и `printf`:

```
print STDERR "help!!!\n";
```

Это аналогично таким английским предложениям, как «Give Gollum the Preciousss»[1], поэтому мы называем это формой *косвенного объекта* (*indirect object*). Ожидается, что инвокант находится в *позиции косвенного объекта* (*indirect object slot*). Когда вы читаете, что встроенной функции, например, `system` или `exec`, что-то передается в ее «позицию косвенного объекта», это означает, что передается этот дополнительный аргумент без запятых в том же месте, в котором он был бы при вызове метода с использованием синтаксиса косвенного объекта.

Форма косвенного объекта даже допускает, что *INVOCANT* задается как *BLOCK*, значением которого является объект (ссылка) или класс (пакет). Это позволяет так объединить два приведенные вызова в одной команде:

```
speak { summon Wizard "Gandalf" } "friend";
```

Синтаксическая путаница с косвенными объектами

Один синтаксис часто оказывается более удобочитаемым, чем другой. Синтаксис косвенного объекта часто оказывается менее загроможденным, но страдает некоторыми формами синтаксической неоднозначности. Первая состоит в том, что часть *LIST* вызова косвенного объекта обрабатывается син-

---

[1] Тот, кто произносит эту фразу (а именно Gollum), говорит о себе в третьем лице. – *Примеч. ред.*

таксическим анализатором так же, как любой другой списковый оператор. Поэтому скобки в:

```
enchant $sword ($pips + 2) * $cost;
```

считаются содержащими аргументы, независимо от того, что следует за ними. В результате это эквивалентно следующему:

```
($sword->enchant($pips + 2)) * $cost;
```

Скорее всего, здесь выполняется не то, что нам нужно: `enchant` вызывается только с `$pips + 2`, а значение, возвращаемое методом, умножается затем на `$cost`. Как и с другими списковыми операторами, нужно также проявлять осторожность в отношении старшинства `&&` и `||` относительно `and` и `or`.

Например,

```
name $sword $oldname || "Glamdring";   # здесь нельзя использовать "or"!
```

становится желаемым:

```
$sword->name($oldname || "Glamdring");
```

но:

```
speak $mage "friend" && enter();        # здесь должно быть "and"!
```

становится двусмысленным:

```
$mage->speak("friend" && enter());
```

что можно исправить, переписав в одной из следующих эквивалентных форм:

```
enter() if $mage->speak("friend");
$mage->speak("friend") && enter();
speak $mage "friend" and enter();
```

Вторая синтаксическая неудачность формы косвенного объекта состоит в том, что в ней *INVOCANT* может быть только именем, скалярной переменной без индекса или блоком.[1] Анализатор получает *INVOCANT* в свое распоряжение, как только встречает что-либо из перечисленного, поэтому он начинает искать список *LIST*. Значит, вызовы:

```
move $party->{LEADER};              # вероятно, ошибочно!
move $riders[$i];                   # вероятно, ошибочно!
```

на практике анализируются как:

```
$party->move->{LEADER};
$riders->move([$i]);
```

[1] Внимательные читатели вспомнят, что это в точности тот список синтаксических элементов, которые допускаются после разыменовывающего символа при разыменовании переменных, например, `@ary`, `@$aryref` или `@{$aryref}`.

хотя, на самом деле, вероятно, требовалось:

```
$party->{LEADER}->move;
$riders[$i]->move;
```

Анализатор лишь чуть-чуть заглядывает вперед, чтобы найти инвокант для косвенного объекта, даже ближе, чем он стал бы искать унарный оператор. Таких странностей не возникает в первом варианте обозначений, поэтому вы, возможно, предпочтете стрелу в качестве оружия.

Даже в английском языке есть аналогичная проблема. Возьмите команду: «Throw your cat out the window a toy mouse to play with». Если слишком поспешно анализировать это предложение, то в результате в окно полетит кошка, а не мышка (если только вы не заметите, что кошка уже за окном).[1] Как и в Perl, в английском есть два варианта синтаксиса для обозначения агента: «Throw your cat the mouse» и «Throw the mouse to your cat». Иногда более длинная форма оказывается понятнее и проще, а иногда – более короткая. В Perl, по крайней мере, требуется заключить сложный косвенный объект в скобки.

Классы, цитирующие имя пакета

Последняя синтаксическая двусмысленность вызова методов в стиле косвенных объектов состоит в том, что они могут вообще не быть опознаны как обращение к методу, если в текущем пакете есть подпрограмма с таким же именем, как у метода. В случае вызова метода класса, имя пакета которого совпадает с именем инвоканта, существует способ разрешения этой неопределенности при сохранении синтаксиса косвенного объекта: добавить удвоенное двоеточие к имени класса, цитируя (package-quote), тем самым, имя пакета:

```
$obj = method CLASS::;    # навязывает "CLASS"->method
```

Это важно, потому что обычная запись:

```
$obj = new CLASS;         # может не быть воспринято как метод
```

не всегда ведет себя так, как нужно, если в текущем пакете есть подпрограмма с именем `new` или *`CLASS`*. Даже если старательно использовать для вызова метода стрелку, а не косвенный объект, в редких случаях все же могут возникать проблемы. Ценой помех, вызываемых лишними знаками пунктуации, обозначение *`CLASS`*`::` гарантирует правильный анализ в Perl вызова метода. Первые два примера из приведенных ниже не всегда анализируются одинаково, в отличие от следующих двух:

---

[1] На русском языке команда могла бы выглядеть так: «Бросьте кошке за окном игрушечную мышку, пусть позабавится». Естественно, в основном тексте речь идет об оригинале. Читая перевод команды, можно только сделать вид, что не очень понятно, кто же будет забавляться. – *Примеч. ред.*

```
$obj = new ElvenRing;              # может быть new("ElvenRing")
                                   # или даже new(ElvenRing())
$obj = ElvenRing->new;             # could be ElvenRing()->new()

$obj = new ElvenRing::;            # всегда "ElvenRing"->new()
$obj = ElvenRing::->new;           # всегда "ElvenRing"->new()
```

Эту нотацию можно сделать симпатичнее путем некоторого выравнивания:

```
$obj = new ElvenRing::
            name    => "Narya",
            owner   => "Gandalf",
            domain  => "fire",
            stone   => "ruby";
```

Все еще можно сказать: «Фу!», глядя на это удвоенное двоеточие, поэтому сообщим, что всегда можно обойтись простым именем класса, если соблюдаются два условия. Во-первых, в классе нет подпрограммы, имя которой совпадает с именем класса. (Если следовать соглашению, что имена подпрограмм, как в `new`, начинаются на нижнем регистре, а имена классов, как в `ElvenRing`, начинаются на верхнем регистре, то это не составляет проблемы.) Во-вторых, класс загружен с помощью одной из двух команд:

```
use ElvenRing;
require ElvenRing;
```

Благодаря наличию любого из этих объявлений Perl знает, что `ElvenRing` представляет собой имя модуля, и любое голое имя вроде `new` перед именем класса `ElvenRing` будет интерпретироваться как вызов метода, даже если вы определили собственную подпрограмму `new` в текущем пакете. Обычно проблем с косвенными классами не возникает, если только не пытаться втиснуть много классов в один файл, из-за чего Perl может не знать, что некоторое имя пакета должно быть именем класса. Те, кто дает своим подпрограммам имена типа `ModuleNames`, тоже в итоге бывают огорчены.

Создание объектов

Все объекты являются ссылками, но не все ссылки являются объектами. Ссылка не сможет действовать как объект, пока ее предмет не будет помечен особым образом, который сообщит Perl о том, к какому пакету он принадлежит. Действие, состоящее в пометке предмета ссылки именем пакета – и, следовательно, класса, т. к. класс – это просто пакет, – состоит в применении к ней функции `bless` и называется *blessing* («благословение»). Можете считать, что при этом ссылка превращается в объект, хотя точнее было бы сказать, что ссылка превращается в ссылку на объект.

Функция `bless` принимает один или два аргумента. Первый аргумент является ссылкой, а второй – пакетом, в котором находится объект ссылки. Если второй аргумент опущен, используется текущий пакет.

```
$obj = { };                 # Получить ссылку на анонимный хеш.
bless($obj);                # Сделать хеш объектом в текущем пакете.
bless($obj, "Critter");     # Сделать хеш объектом в классе Critter.
```

Здесь мы использовали ссылку на анонимный хеш, который обычно выступает в объектах как структура данных. Хеши являются очень гибкой конструкцией. Но хотелось бы подчеркнуть, что можно применить `bless` к ссылке, указывающей на любой элемент, на который можно ссылаться в Perl, в том числе скаляры, массивы, подпрограммы и typeglob. Можно применить `bless` даже к хешу таблицы имен пакета, если для этого найдется причина. (И даже если не найдется.) Объектная ориентированность в Perl совершенно ортогональна структуре данных.

После того как предмет ссылки «благословлен», применение встроенной функции `ref` к ссылке на него возвращает не встроенный тип, а имя его класса, например `HASH`. Если вам нужен встроенный тип, используйте функцию `reftype` из модуля `attributes`. См. описание `use attributes` в главе 31 «Модули прагм».

А вот как создается объект. Берется какая-то ссылка, ей придается класс путем его «благословления» в пакете – и все. Это все, что нужно сделать, если создается минимальный класс. Если класс лишь используется, то нужно еще меньше, потому что разработчик класса спрятал `bless` в методе, называемом конструктором, который создает и возвращает экземпляры класса. Поскольку `bless` возвращает свой первый аргумент, обычный конструктор может быть очень прост:

```
package Critter;
sub spawn { bless {}; }
```

Или, если расписать все несколько подробнее:

```
package Critter;
sub spawn {
    my     $self = {};        # Ссылка на пустой анонимный хеш
    bless  $self, "Critter"; # Сделать этот хеш объектом Critter
    return $self;             # Возвратить только что созданный Critter
}
```

Имея такое определение, можно создавать объект `Critter`:

```
$pet = Critter->spawn;
```

Наследуемые конструкторы

Как и все методы, конструктор – это просто подпрограмма, но мы не вызываем его как подпрограмму. Мы всегда вызываем его как метод – метод класса, в данном случае, потому что инвокантом является имя пакета. Вызовы методов отличаются от вызовов обычных подпрограмм в двух отношениях. Во-первых, они получают дополнительный аргумент, о чем говорилось выше.

Во-вторых, они подчиняются наследованию, разрешающему одному классу использовать методы другого.

Механику наследования мы обсудим более строго в следующем разделе, а пока несколько простых примеров его действия должны помочь вам в разработке собственных конструкторов. Допустим, что у нас есть класс Spider, наследующий методы класса Critter (живое существо). В частности, пусть в классе Spider нет собственного метода spawn. Применяется следующее соответствие:

Вызов метода	Результирующий вызов подпрограммы
Critter->spawn()	Critter::spawn("Critter")
Spider->spawn()	Critter::spawn("Spider")

В обоих случаях вызывается одна и та же подпрограмма, но с разными аргументами. Обратите внимание на то, что наш конструктор spawn, приведенный выше, совершенно игнорировал свой аргумент, и это значит, что наш объект Spider был некорректно «благословлен» в класс Critter. Более правильный конструктор должен передать bless имя пакета (полученное в качестве первого аргумента):

```
sub spawn {
    my $class =  shift;       # Сохранить имя пакета
    my $self  =  { };
    bless($self, $class);     # «Благословить» ссылку в этот пакет
    return $self;
}
```

Теперь в обоих случаях можно использовать одну и ту же подпрограмму:

```
$vermin = Critter->spawn;
$shelob = Spider->spawn;
```

И каждый объект будет принадлежать правильному классу. Теперь это работает даже косвенным образом:

```
$type  = "Spider";
$shelob = $type->spawn;          # То же, что и "Spider"->spawn
```

Это по-прежнему метод класса, а не экземпляра, потому что его инвокант содержит строку, а не ссылку.

Если бы $type была объектом, а не именем класса, то приведенное определение конструктора было бы неработоспособно, потому что для bless требуется имя класса. Но для многих классов есть смысл в использовании существующего объекта в качестве шаблона для создания нового объекта. В таких случаях можно разрабатывать такие конструкторы, которые работают и с объектами, и с именами классов:

```
sub spawn {
    my $invocant = shift;
```

```
    my $class    = ref($invocant) || $invocant;  # Объект или имя класса
    my $self     = { };
    bless($self, $class);
    return $self;
}
```

Инициализаторы

Большинство объектов хранят внутреннюю информацию, которая косвенно обрабатывается методами объекта. До сих пор все наши конструкторы создавали пустые хеши, но нет причин оставлять их пустыми. Например, можно сделать так, что конструктор будет принимать дополнительные аргументы и хранить их в хеше в виде пар ключ/значение. В литературе по ООП такие данные часто называются *свойствами*, *атрибутами*, *методами доступа*, *данными-членами*, *данными экземпляра* и *переменными экземпляра* (*properties*, *attributes*, *accessors*, *member data*, *instance data*, *instance variables*). В разделе «Переменные экземпляра» далее в этой главе атрибуты обсуждаются более подробно.

Представим себе класс `Horse` (лошадь) с атрибутами «name» (кличка) и «color» (масть):

```
$steed = Horse->new(name => "Shadowfax", color => "white");
```

Если объект реализован как ссылка на хеш, пары ключ/значение можно непосредственно интерполировать в хеш, как только инвокант удален из списка аргументов:

```
sub new {
    my $invocant = shift;
    my $class = ref($invocant) || $invocant;
    my $self = { @_ };          # Оставшиеся аргументы становятся атрибутами
    bless($self, $class);       # «Благословление» в объекты
    return $self;
}
```

На этот раз в качестве конструктора класса мы использовали метод с именем `new`, что может создать у программистов C++ ощущение понимания происходящего. Но Perl не рассматривает слово «`new`», как имеющее особый смысл; вы можете называть свои конструкторы как угодно. Любой метод, создающий и возвращающий объект, является конструктором de facto. В целом, мы советуем давать конструкторам осмысленные в контексте решаемой задачи имена. Например, конструкторы в модуле `Tk` имеют имена, соответствующие создаваемым ими графическим объектам. В модуле `DBI` конструктор с именем `connect` возвращает объект – дескриптор базы данных, а другой конструктор, с именем `prepare`, вызывается как метод экземпляра и возвращает объект – дескриптор команды. Но если нет подходящего в данном контексте имени конструктора, то `new` является не самым ужасным выбором. С другой стороны, не так уж плохо выбрать какое-то случайное имя, которое

заставит людей прочесть контракт с интерфейсом (т. е. документацию по классу), прежде чем они станут использовать его конструкторы.

В ходе дальнейшей разработки можете оснастить свой конструктор парами ключ/значение по умолчанию, которые пользователь может в дальнейшем переопределить с помощью переданных аргументов:

```
sub new {
    my $invocant = shift;
    my $class   = ref($invocant) || $invocant;
    my $self = {
        color  => "гнедой",
        legs   => 4,
        owner  => undef,
        @_,                 # Заместить прежние атрибуты
    };
    return bless $self, $class;
}

$ed       = Horse->new;                     # 4-ногая гнедая лошадь
$stallion = Horse->new(color => "черный");  # 4-ногая вороная лошадь
```

Конструктор `Horse` игнорирует имеющиеся у его инвоканта атрибуты при использовании в качестве метода экземпляра. Можно было бы создать второй конструктор, предназначенный для вызова в качестве метода экземпляра, и сделать так, чтобы значения вызывающего объекта принимались как значения по умолчанию для нового объекта:

```
$steed  = Horse->new(color => "серовато-коричневый");
$foal   = $steed->clone(owner => "EquuGen Guild, Ltd.");

sub clone {
    my $model = shift;
    my $self  = $model->new(%$model, @_);
    return $self;     # Ранее «благословленный» с помощью ->new
}
```

(Можно было бы поместить эти возможности прямо в `new`, но тогда имя не вполне соответствовало бы функции.)

Обратите внимание, что даже в конструкторе `clone` мы не кодируем жестко `Horse` — имя класса. Мы делаем так, что исходный объект, чем бы он ни был, вызывает свой собственный метод `new`. Если бы вместо `$model->new` мы написали `Horse->new`, то этот класс не позволил бы наследовать классам `Зебра` или `Единорог`. Не хотелось бы, клонируя Пегаса, получить лишь лошадь другой масти.

Иногда, однако, возникает противоположная проблема: вместо того, чтобы делить один конструктор между несколькими классами, желательно иметь несколько конструкторов для объекта одного класса. Это случается, когда конструктор хочет вызвать конструктор базового класса, чтобы тот выполнил часть работы. Perl не производит иерархического создания объектов. Это значит, что Perl не станет автоматически вызывать конструкторы (или

деструкторы) для каких-либо базовых классов создаваемого объекта, поэтому вашему конструктору придется сделать это самостоятельно, а затем добавить дополнительные атрибуты, необходимые производному классу. Ситуация отчасти сходна с программой clone, за исключением того, что вместо копирования существующего объекта в новый объект необходимо вызвать конструктор своего базового класса, а затем превратить новый базовый объект в новый производный объект.

Наследование классов

Как и вообще в системе объектов Perl, наследование одного класса другим не требует ввода в язык особого синтаксиса. При вызове метода, для которого Perl не находит подпрограммы в пакете инвоканта, исследуется массив @ISA[1] этого пакета. Вот как Perl реализует наследование: каждый элемент массива @ISA данного пакета содержит имя другого пакета, в котором производится поиск отсутствующих методов. Например, в следующем примере класс Horse становится подклассом класса Critter. (Мы объявляем @ISA как our, потому что он должен быть переменной пакета, а не лексической переменной, объявляемой через my.)

```
package Horse;
our @ISA = "Critter";
```

Теперь вы должны быть в состоянии использовать класс или объект Horse везде, где встречался класс Critter. Если ваш новый класс пройдет эту проверку на пустой подкласс, то вы будете знать, что Critter является правильным базовым классом, пригодным для наследования.

Допустим, что в $steed находится объект Horse, и вы вызываете с ним метод move:

```
$steed->move(10);
```

Поскольку $steed является объектом Horse, Perl сначала ищет для этого метода подпрограмму Horse::move. Если такой не найден, то Perl не возбуждает исключительную ситуацию на этапе исполнения, а обращается к первому элементу массива @Horse::ISA, который отправляет его в пакет Critter в поисках Critter::move. Если и эта подпрограмма не найдена, а в Critter есть *собственный* массив @Critter::ISA, то и в нем начнется поиск имени пакета-предка, который мог бы предоставить метод move, и так движение по иерархии наследования будет продолжаться до достижения пакета, в котором нет @ISA.

Описанная только что ситуация называется *одиночным наследованием* (*single inheritance*), при этом у каждого класса есть только один родитель. Такое наследование подобно связанному списку родственных пакетов. Perl поддерживает также *множественное наследование* (*multiple inheritance*); нужно

[1] Произносится «изэ», как в «is a».

просто добавить еще пакеты в @ISA класса. Такого рода наследование действует скорее как структура данных в виде дерева, потому что у каждого пакета может иметься более одного непосредственного родителя. Некоторые находят, что это более сексуально.

При вызове метода *methname* с инвокантом типа *classname* Perl опробует шесть разных способов, чтобы найти подпрограмму, которую нужно использовать:

1. Сначала Perl ищет в собственном пакете инвоканта подпрограмму с именем *classname*::*methname*. Если это не удается, вступает в действие наследование, и мы переходим к шагу 2.
2. Затем Perl проверяет методы, унаследованные от базовых классов, просматривая все пакеты *parent*, которые перечислены в @*classname*::ISA, в поисках подпрограммы *parent*::*methname*. Поиск ведется слева направо, рекурсивно и сначала вглубь. Рекурсия обеспечивает проведение поиска во всех классах – прародителей, прапрародителей, прапрапрародителей и т. д.
3. Если это не помогло, Perl ищет подпрограмму с именем UNIVERSAL::*methname*.
4. В этом месте Perl отчаивается найти *methname* и начинает искать AUTOLOAD. Сначала он ищет подпрограмму с именем *classname*::AUTOLOAD.
5. Если этот поиск неудачен, Perl ищет во всех пакетах *parent*, перечисленных в @*classname*::ISA, какую-нибудь подпрограмму *parent*::AUTOLOAD. Поиск опять ведется слева направо, рекурсивно и сначала вглубь.
6. Наконец, Perl ищет подпрограмму с именем UNIVERSAL::AUTOLOAD.

Perl прекращает поиск после первой же удачи и вызывает соответствующую подпрограмму. Если подпрограмма не найдена, возбуждается исключительная ситуация (которую вы будете наблюдать нередко):

```
Can't locate object method "methname" via package "classname"
(Не могу найти объектный метод "methname" через пакет "classname")
```

Если ваша версия Perl собрана с поддержкой отладки при помощи опции компилятора -DDEBUGGING, то при задании Perl ключа -Do можно увидеть прохождение всех этих шагов во время разрешения вызова метода.

По мере изложения мы будем обсуждать механизм наследования более подробно.

Наследование через @ISA

Если массив @ISA содержит больше одного имени пакета, то поиск в пакетах осуществляется в порядке слева направо. Поиск ведется сначала вглубь, поэтому, если у вас есть класс Mule (мул), так подготовленный для наследования:

```
package Mule;
our @ISA = ("Horse", "Donkey");
```

то Perl ищет методы, отсутствующие в Mule, сначала в Horse (и всех его предках, например, Critter) и только потом переходит к поиску в Donkey (осел) и его предках.

Если отсутствующий метод найден в базовом классе, то Perl внутренне кэширует этот метод в текущем пакете для повышения эффективности, поэтому когда ему потребуется найти данный метод в следующий раз, не надо будет ходить далеко. Изменение @ISA или определение новых методов делает кэш недействительным и заставляет Perl снова выполнять поиск.

Когда Perl ищет метод, он проверяет, не создана ли циклическая иерархия наследования. Это может произойти, если два класса наследуют друг другу, в том числе косвенно, через другие классы. Попытка стать собственным прадедом парадоксальна даже для Perl, поэтому она возбуждает исключительную ситуацию. Однако Perl не считает ошибкой наследование от нескольких классов, имеющих общих предков, что напоминает брак между кузенами. Иерархия наследования просто перестает быть деревом и начинает выглядеть как направленный ациклический граф. Perl это не заботит – до тех пор, пока граф действительно остается ациклическим.

Когда вы устанавливаете значения @ISA, присвоение происходит обычно на этапе исполнения, поэтому, если не принять мер предосторожности, код в блоках BEGIN, CHECK и INIT не будет иметь возможность использовать иерархию наследования. Одним из средств предосторожности (или удобства) является прагма use base, которая позволяет выполнять require для классов и добавлять их в @ISA на этапе компиляции. Вот как можно ее использовать:

```
package Mule;
use base ("Horse", "Donkey");   # объявление надклассов
```

Это сокращенная запись для следующего:

```
package Mule;
BEGIN {
    our @ISA = ("Horse", "Donkey");
    require Horse;
    require Donkey;
}
```

за исключением того, что use base учитывает также объявления use fields.

Некоторых удивляет, что включение класса в @ISA не влечет require соответствующего модуля. Это связано с тем, что система классов Perl в значительной мере ортогональна его системе модулей. В одном файле может содержаться много классов (поскольку они являются просто пакетами), а один пакет может указываться в нескольких файлах. Но в наиболее обычной ситуации, когда один пакет, один класс, один модуль и один файл в конечном итоге оказываются весьма взаимозаменяемыми – если сильно прищуриться, прагма use base предлагает синтаксис объявления, который устанавливает наследование, загружает файлы модулей и размещает любые объявленные поля базового класса. Это одна из удобных диагоналей, о которых мы говорим.

Дополнительные сведения можно найти в описаниях `use base` и `use fields` в главе 31 «Модули прагм».

Доступ к замещенным методам

Когда класс определяет метод, то эта подпрограмма замещает одноименные методы во всех базовых классах. Представьте себе, что у вас есть объект `Mule` (производный от класса `Horse` и класса `Donkey`) и вы решаете вызвать метод `breed` (скрещивание) вашего объекта. Хотя у родительских классов есть собственные методы `breed`, разработчик класса `Mule` заместил их, снабдив класс `Mule` собственным методом `breed`. Это значит, что следующее скрещивание вряд ли будет продуктивным:

```
$stallion = Horse->new(gender => "male");
$molly = Mule->new(gender => "female");
$colt = $molly->breed($stallion);
```

Теперь допустим, что чудеса генной инженерии позволили вам решить известную проблему стерильности мула, и вы хотите перескочить через нежизнеспособный метод `Mule::breed`. Можно вызвать свой метод как обычную подпрограмму, обеспечив явную передачу инвоканта:

```
$colt = Horse::breed($molly, $stallion);
```

Однако это отход от наследования, что почти всегда является неверным путем. Вполне можно представить себе, что подпрограммы `Horse::breed` не существует, потому что `Horse` и `Donkey` получают это свое поведение от общего родительского класса с именем `Equine`. Если, с другой стороны, необходимо указать Perl, чтобы он *начал* поиск метода в определенном классе, выберите обычный метод вызова, но квалифицируйте имя метода именем класса:

```
$colt = $molly->Horse::breed($stallion);
```

Иногда требуется, чтобы метод производного класса действовал как оболочка вокруг некоторого метода базового класса. Метод в производном классе может сам вызвать метод базового класса, добавив собственные действия до или после этого вызова. *Можно* было бы использовать только что продемонстрированный способ записи для указания того, в каком классе должен начаться поиск. Но в большинстве случаев применения замещенных методов нежелательно, чтобы было необходимо знать или указывать родительский класс, замещенный метод которого должен выполняться.

В таких случаях удобно использовать псевдокласс `SUPER`. Он позволяет вызывать замещенный метод базового класса, не указывая, в каком классе этот метод определен.[1] Следующая подпрограмма ведет поиск в `@ISA` текущего пакета, не заставляя вас указывать конкретные классы:

---

[1] Не надо путать это с механизмом, о котором говорилось в главе 11 «Модули» для замещения встроенных функций Perl, не являющихся методами объектов и потому не замещаемых путем наследования. Замещенные функции вызываются через псевдопакет `CORE`, а не псевдопакет `SUPER`.

```
package Mule;
our @ISA = qw(Horse Donkey);
sub kick {
    my $self = shift;
    print "Мул лягается!\n";
    $self->SUPER::kick(@_);
}
```

Псевдопакет SUPER имеет значение только в случае применения *внутри* метода. Тот, кто реализует класс, может включить SUPER в свой код, но тот, кто просто использует объекты класса – нет.

SUPER не всегда работает так, как вам хотелось бы, если происходит множественное наследование. Как можно ожидать, он использует @ISA как обычный механизм наследования: слева направо, рекурсивно, сначала вглубь. Если бы метод speak был у обоих классов, Horse и Donkey, и вы предпочли бы метод Donkey, то пришлось бы явно указать этот родительский класс:

```
sub speak {
    my $self = shift;
    print "Мул разговаривает!\n";
    $self->Donkey::speak(@_);
}
```

Более сложные подходы к ситуациям с множественным наследованием могут разрабатываться с помощью метода UNIVERSAL::can, который описывается в следующем разделе. Можете также воспользоваться модулем Class::Multimethods из CPAN, в котором есть много усовершенствованных решений, в том числе поиск ближайшего соответствия вместо самого левого.

В любом месте кода Perl известно, какой пакет является текущим, т. е. определен последней командой package. Метод SUPER обращается к @ISA только того пакета, в который был скомпилирован вызов SUPER. Его не интересуют ни класс инвоканта, ни пакет вызываемой подпрограммы. Это может послужить источником проблем, если попытаться определить методы в другом классе путем лишь манипуляций с именем метода:

```
package Bird;
use Dragonfly;
sub Dragonfly::divebomb { shift->SUPER::divebomb(@_) }
```

К несчастью, при этом вызывается надкласс Bird, а не Dragonfly. Чтобы осуществить то, что вы хотите, нужно явно перейти в соответствующий пакет, и тогда SUPER правильно скомпилируется:

```
package Bird;
use Dragonfly;
{
    package Dragonfly;
    sub divebomb { shift->SUPER::divebomb(@_) }
}
```

Как показывает этот пример, не требуется редактировать файл модуля, чтобы добавить новые методы к существующему классу. Поскольку класс является просто пакетом, а метод – просто подпрограммой, нужно лишь определить функцию в нужном пакете, как мы здесь и сделали, и в классе внезапно появится новый метод. Никакого наследования не требуется. Значение имеет только пакет, а т. к. пакеты являются глобальными, то доступ к любому пакету может быть осуществлен из любого места программы. (Кстати, мы уже говорили, что собираемся на следующей неделе поставить у вас в комнате джакузи?)

UNIVERSAL: первичный класс предков

Если после поиска в классе инвоканта и рекурсивного поиска во всех классах, являющихся его предками, не найдено определения метода с нужным именем, производится еще одна проверка метода с этим именем в особом предопределенном классе с именем `UNIVERSAL`. Этот пакет никогда не указывается в `@ISA`, но проверка в нем всегда производится, если поиск по `@ISA` оказывается неудачным. Можно считать `UNIVERSAL` изначальным прародителем, от которого неявным образом производятся все классы.

Ниже описаны предопределенные методы, имеющиеся в классе `UNIVERSAL`, а следовательно и во всех классах. Все они работают независимо от того, вызываются ли они как методы класса или как методы объекта.

INVOCANT`->isa(`*CLASS*`)`

Метод `isa` возвращает true, если классом ивоканта *INVOCANT* является *CLASS* или любой класс, наследующий *CLASS*. При этом *CLASS* может быть не именем пакета, а одним из встроенных типов, например, `"HASH"` или `"ARRAY"`. (Однако выяснение точного типа не сулит ничего хорошего инкапсуляции и полиморфизму. Чтобы получить правильный метод, следует полагаться на диспетчеризацию методов.)

```
use FileHandle;
if (FileHandle->isa("Exporter")) {
    print "FileHandle есть Exporter.\n";
}

$fh = FileHandle->new();
if ($fh->isa("IO::Handle")) {
    print "\$fh есть какой-то объект IO.\n";
} if ($fh->isa("GLOB")) {
    print "\$fh есть ссылка на GLOB.\n";
}
```

INVOCANT`->can(`*METHOD*`)`

Метод `can` возвращает ссылку на подпрограмму, которая была бы вызвана при применении *METHOD* к *INVOCANT*. Если такая подпрограмма не найдена, возвращается `undef`.

```
if ($invocant->can("copy")) {
    print "Наш инвокант умеет копировать.\n";
}
```

Это можно использовать, чтобы вызывать метод только при условии его существования:

```
$obj->snarl if $obj->can("snarl");
```

При множественном наследовании это позволяет методу вызвать все замещенные методы базового класса, а не только самый левый:

```
sub snarl {
    my $self = shift;
    print "Ворчащие (snarling): @_\n";
    my %seen;
    for my $parent (@ISA) {
        if (my $code = $parent->can("snarl")) {
            $self->$code(@_) unless $seen{$code}++;
        }
    }
}
```

Мы используем хеш `%seen`, чтобы следить за тем, какие подпрограммы мы уже вызывали, и избежать таким образом повторного вызова одной и той же подпрограммы. Это могло бы произойти, если бы у нескольких родительских классов был общий предок.

Методы, которые запускают `AUTOLOAD` (описываемый в следующем разделе), учитываются неточно, за исключением случая, когда в пакете объявлены (но не определены) подпрограммы, автоматической загрузки которых он желает.

INVOCANT`->VERSION(`*NEED*`)`

Метод `VERSION` возвращает значение номера версии класса инвоканта *INVOCANT*, хранящееся в переменной пакета `$VERSION`. Если передается аргумент *NEED*, то метод проверяет, не меньше ли номер текущей версии, чем *NEED*, и если это так, возбуждается исключительная ситуация. Этот метод использует `use` для определения того, достаточно ли свеж модуль.

```
use Thread 1.0;   # вызывает Thread->VERSION(1.0)
print "Выполняется Thread-версия ", Thread->VERSION, ".\n";
```

Вы можете создать собственный метод `VERSION` для замещения метода в `UNIVERSAL`. Однако при этом все классы, производные от вашего, тоже будут использовать замещающий метод. Для того чтобы это не происходило, метод должен отправлять запросы версии, исходящие от других классов, снова в `UNIVERSAL`.

Находящиеся в `UNIVERSAL` методы представляют собой встроенные подпрограммы Perl, которые можно вызывать, полностью квалифицируя и передавая два аргумента, например, `UNIVERSAL::isa($formobj, "HASH")`. (Делать так,

однако, не рекомендуется, поскольку обычно ответ на ваш вопрос может дать can.)

Можно добавлять в класс UNIVERSAL собственные методы. (Конечно, следует проявлять осторожность: можно запутать кого-либо, кто рассчитывает на *отсутствие* метода с определенным вами именем и осуществление его автозагрузки из какого-либо другого места.) Сейчас мы создадим метод copy, который могут использовать объекты всех классов, не определившие собственный такой метод. (При этом мы демонстрируем неудачу метода, если он вызван с классом, а не объектом.)

```
use Data::Dumper;
use Carp;
sub UNIVERSAL::copy {
    my $self = shift;
    if (ref $self) {
        return eval Dumper($self);  # ссылки на CODE не допускаются
    } else {
        confess "UNIVERSAL::copy не может копировать класс $self";
    }
}
```

Эта стратегия Data::Dumper не работает, если объект содержит ссылки на подпрограммы, потому что они не могут быть правильно воспроизведены. Даже если доступен исходный код, лексические привязки будут утеряны.

Автозагрузка методов

Обычно при обращении к неопределенной подпрограмме в пакете, который определяет подпрограмму AUTOLOAD, вместо возбуждения исключительной ситуации вызывается подпрограмма AUTOLOAD (см. раздел «Автозагрузка» главы 10 «Пакеты»). Для методов это действует несколько иначе. Если обычный поиск метода (через класс, его предков и, наконец, UNIVERSAL) не может найти соответствия, та же последовательность поиска выполняется снова, но на этот раз выполняется поиск подпрограммы AUTOLOAD. Если она найдена, то вызывается в качестве метода, при этом переменная пакета $AUTOLOAD устанавливается равной полностью квалифицированному имени подпрограммы, по поручению которой была вызвана AUTOLOAD.

При автозагрузке методов следует проявлять некоторую осторожность. Во-первых, подпрограмма AUTOLOAD должна немедленно осуществить возврат, если вызвана по поручению метода с именем DESTROY, если только вашей целью не была эмуляция DESTROY, имеющего в Perl особое значение, как описано в разделе «Деструкторы экземпляров» далее в этой главе.

```
sub AUTOLOAD {
    return if our $AUTOLOAD =~ /::DESTROY$/;
    ...
}
```

Во-вторых, если класс предоставляет страховку в виде AUTOLOAD, то невозможно использовать UNIVERSAL::can с именем метода, чтобы проверить, безопасно ли его вызывать. Проверять AUTOLOAD нужно отдельно:

```
if ($obj->can("methname") || $obj->can("AUTOLOAD")) {
    $obj->methname();
}
```

Наконец, когда при множественном наследовании класс наследует двум или более классам, в каждом из которых есть AUTOLOAD, запускается только самый левый из них, т. к. Perl останавливается, найдя первый AUTOLOAD.

Последние две странности легко обойти, объявив подпрограммы в пакете, чей AUTOLOAD должен справиться с этими методами. Можно сделать это с помощью отдельных объявлений:

```
package Goblin;
sub kick;
sub bite;
sub scratch;
```

или с помощью прагмы use subs, что более удобно, если надо объявить много методов:

```
package Goblin;
use subs qw(kick bite scratch);
```

Несмотря на то что эти подпрограммы лишь объявлены, но не определены, системе этого достаточно, чтобы считать их существующими. Они видны при проверке UNIVERSAL::can и, что более важно, на шаге 2 поиска метода, который никогда не перейдет к шагу 3, не говоря уже о шаге 4.

«Но послушайте»,– воскликнете вы,– «они же вызывают AUTOLOAD, не так ли» В конечном счете, да, но механизм этого иной. Найдя на шаге 2 заглушку метода, Perl пытается вызвать его. При обнаружении, что метод – не то, чем ему полагается быть, снова начинается поиск метода AUTOLOAD, но на этот раз поиск начинается в классе, содержащем заглушку, что ограничивает поиск этим классом и его предками (и UNIVERSAL). Благодаря этому Perl находит для запуска правильный AUTOLOAD и умеет игнорировать методы AUTOLOAD, находящиеся в неверной части исходного дерева наследования.

Закрытые методы

Есть такой способ вызова метода, при котором Perl вообще игнорирует наследование. Если вместо литерального имени метода задать простую скалярную переменную, содержащую ссылку на подпрограмму, то подпрограмма вызывается непосредственно. В описании UNIVERSAL->can в предыдущем разделе последний пример вызывает все замещенные методы с помощью ссылки на подпрограмму, а не ее имени.

Интересная сторона такого режима состоит в том, что его можно применять для реализации вызовов закрытых методов. Если поместить класс в модуль,

то можно использовать лексическую область видимости файла для создания закрытых методов. Сначала сохраним анонимную подпрограмму в лексической переменной с файловой областью видимости:

```
# объявление закрытого метода
my $secret_door = sub {
    my $self = shift;
    ...
};
```

Далее переменную можно использовать в этом файле, как если бы она содержала имя метода. Замыкание будет вызвано непосредственно, невзирая на наследование. Как и для любого другого метода, в качестве дополнительного аргумента передается инвокант.

```
sub knock {
    my $self = shift;
    if ($self->{knocked}++ > 5) {
        $self->$secret_door();
    }
}
```

Это позволяет собственным подпрограммам файла (методам класса) вызывать метод, к которому код вне лексической области видимости не имеет доступа.

Деструкторы экземпляров

Как и для любого другого объекта ссылки в Perl, как только исчезает последняя ссылка на объект, его память неявно освобождается для повторного использования. Объекты позволяют перехватить управление непосредственно перед тем, как это произойдет, путем определения подпрограммы DESTROY в пакете класса. Этот метод автоматически запускается в нужный момент, и в качестве единственного объекта ему передается объект, который вот-вот должен быть уничтожен.

Деструкторы редко требуются в Perl, потому что управление памятью осуществляется автоматически. Однако структура некоторых объектов может выходить за пределы памяти, и о ней желательно было бы позаботиться; например, это касается дескрипторов файлов и соединений с базами данных.

```
package MailNotify;
sub DESTROY {
    my $self = shift;
    my $fh   = $self->{mailhandle};
    my $id   = $self->{name};
    print $fh "\n$id отключается в " . localtime() . "\n";
    close $fh;  # закрыть канал обработчика почты
}
```

Так же как для создания объекта Perl достаточно единственного метода, даже если класс конструктора наследует одному или нескольким другим классам, Perl использует только один метод DESTROY для каждого объекта, невзирая на наследование. Иными словами, Perl не станет выполнять для вас иерархическое уничтожение. Если ваш класс замещает деструктор надкласса, то вашему методу DESTROY может понадобиться вызвать метод DESTROY для всех соответствующих базовых классов:

```
sub DESTROY {
    my $self = shift;
    # Проверка замещения деструктора...
    $self->SUPER::DESTROY if $self->can("SUPER::DESTROY");
    # теперь делайте свои собственные дела
}
```

Это относится только к производным классам; объект, который просто *содержится* в текущем объекте – как, например, одно значение в более крупном хеше – будет освобожден и уничтожен автоматически. Это одна из причин, по которым вложенность путем простого агрегирования (иногда называемая отношением «has-a» – «имеет») часто является более беспримесной и понятной, чем наследование (отношение «is-a» – «является»). Иными словами, часто нужно лишь поместить один объект непосредственно внутрь другого, избегая наследования, которое вносит ненужную сложность. Иногда пользователи хватаются за множественное наследование, в то время как вполне достаточно одиночного.

Явное обращение к DESTROY допускается, но редко требуется. Оно может даже оказаться вредным, поскольку повторный запуск деструктора одного и того же объекта может привести к неприятным последствиям.

Сборка мусора методами DESTROY

Как описывалось в разделе «Сборка мусора, циклические и слабые ссылки» главы 8 «Ссылки», переменные, которые ссылаются на самих себя (или несколько ссылок, ссылающихся друг на друга косвенно), не освобождаются до выхода из программы (или встроенного интерпретатора). Для того чтобы раньше возвратить занимаемую ими память, обычно приходится явным образом разорвать ссылку или ослабить ее с помощью модуля WeakRef из CPAN.

Для объектов другим решением может быть создание класса-контейнера, содержащего указатель на структуру данных со ссылками на саму себя. Определите метод DESTROY для класса, содержащего объект, который явно разрушает цикличность в структуре данных, содержащей ссылки на саму себя. Пример такого рода можно найти в главе 13 книги «Perl: библиотека программиста» Т. Кристиансена и Н. Торкингтона («Perl Cookbook») в рецепте 13.13 «Как справиться с циклическими структурами данных».

Когда интерпретатор завершает работу, все его объекты уничтожаются, что важно для многопоточных приложений и приложений со встроенным Perl. Объекты уничтожаются в отдельном проходе, до обычных ссылок. Это дела-

ется для того, чтобы предотвратить использование методами DESTROY уже уничтоженных ссылок. (А также потому, что обыкновенные ссылки попадают в сборку мусора только во встроенных интерпретаторах, поскольку выход (exit) из процесса является очень *быстрым* способом утилизации ссылок. Но при выходе не запускаются деструкторы объектов, поэтому сначала Perl делает это.)

Управление данными экземпляров

Большинство классов создает объекты, которые, по существу, являются просто структурами данных с несколькими внутренними полями данных (переменными экземпляра), для работы с которыми добавлены методы.

Классы Perl наследуют методы, а не данные, но если любой доступ к объекту осуществляется через вызовы методов, то это отлично действует. Если вам требуется наследование данных, его нужно осуществлять через наследование методов. В целом, это не является необходимостью в Perl, поскольку большинство классов хранит данные своих объектов в анонимных хешах. Данные экземпляра объекта хранятся в этом хеше, служащем собственным маленьким пространством имен, которое будут разбирать классы, когда им нужно что-то сделать с объектом. Например, если нужно, чтобы в объекте $city было поле данных с именем elevation, можно просто обратиться к $city>{elevation}. Никаких объявлений не требуется. Но и у методов-оболочек есть свое применение.

Допустим, надо реализовать объект Person. Принято решение, что необходимо поле данных «name», которое, по странному совпадению, будет храниться с ключом name в анонимном хеше, служащем в качестве объекта. Но вы не хотите, чтобы пользователи входили в прямой контакт с данными. Чтобы пожинать плоды инкапсуляции, у пользователей должны быть методы для доступа к этой переменной экземпляра без поднятия завесы абстрактности.

Например, можно создать пару методов доступа:

```
sub get_name {
    my $self = shift;
    return $self->{name};
}

sub set_name {
    my $self      = shift;
    $self->{name} = shift;
}
```

что ведет к использованию такого кода:

```
$him = Person->new();
$him->set_name("Frodo");
$him->set_name( ucfirst($him->get_name) );
```

Можно даже объединить два метода в один:

```
sub name {
    my $self = shift;
    if (@_) { $self->{name} = shift }
    return $self->{name};
}
```

Это ведет к использованию такого кода:

```
$him = Person->new();
$him->name("Frodo");
$him->name( ucfirst($him->name) );
```

Преимуществом написания отдельных функций для каждой переменной экземпляра (для нашего класса `Person` это могут быть name – имя, age – имя, height – имя и т. д.) являются непосредственность, очевидность и гибкость. Недостатком является то, что при возникновении необходимости в новом классе приходится определять один или два почти идентичных метода для каждой переменной экземпляра. На первый взгляд, в этом нет ничего страшного, и никому не возбраняется выбрать тот способ, который больше понравится. Но если отдается предпочтение удобству перед гибкостью, то более предпочтительным может оказаться один из приемов, описанных в следующих разделах.

Обратите внимание на то, что мы будем изменять реализацию, а не интерфейс. Если пользователи вашего класса с уважением относятся к инкапсуляции, вы можете переходить от одной реализации к другой так, что они ничего не заметят. (Родственники из вашего дерева наследования, использующие ваш класс в качестве подкласса или надкласса, могут быть не столь покладисты, поскольку знают вас лучше, чем посторонние.) Если ваши пользователи совали свой нос в частные дела вашего класса, то неминуемо ожидающая их катастрофа – их вина и не ваша забота. Все, что вы можете сделать – это до конца придерживаться контракта, сохраняя интерфейс. Попытка помешать кому бы то ни было сделать что-то хоть сколько-нибудь вредное отнимет все ваше время и силы – и в конце концов все равно провалится.

Договориться с членами семьи сложнее. Если подкласс замещает метод доступа к атрибуту из надкласса, должен он обращаться к тому же полю хеша или нет? Можно привести аргументы в пользу того и другого, в зависимости от природы атрибута. В общем случае в интересах безопасности каждый метод доступа может сделать префиксом имени поля хеша имя своего класса, чтобы подкласс и надкласс имели собственные варианты поля. Некоторые из приведенных ниже примеров, а также стандартный модуль `Struct::Class` используют такую стратегию защиты от подкласса. Можно видеть методы доступа такого типа:

```
sub name {
    my $self = shift;
    my $field = __PACKAGE__ . "::name";
```

```
    if (@_) { $self->{$field} = shift }
    return $self->{$field};
}
```

В каждом из последующих примеров мы создаем простые классы `Person` с полями `name`, `race` и `aliases`, интерфейсы которых идентичны, но реализация совершенно разная. Мы не скажем, который из них нам больше нравится, потому что они все нам нравятся, в зависимости от ситуации. О вкусах не спорят. Кто любит генерала, а кто – свиной хрящик.

Объявление полей с помощью use fields

Объекты не обязательно нужно реализовывать как анонимные хеши. Можно использовать любую ссылку. Например, в случае применения анонимного массива можно организовать такой конструктор:

```
sub new {
    my $invocant = shift;
    my $class = ref($invocant) || $invocant;
    return bless [], $class;
}
```

и такие методы доступа:

```
sub name {
    my $self = shift;
    if (@_) { $self->[0] = shift }
    return $self->[0];
}

sub race {
    my $self = shift;
    if (@_) { $self->[1] = shift }
    return $self->[1];
}

sub aliases {
    my $self = shift;
    if (@_) { $self->[2] = shift }
    return $self->[2];
}
```

Доступ к массивам происходит несколько быстрее, чем к хешам, и они занимают меньше памяти, но пользоваться ими не совсем удобно. Приходится следить за номерами индексов (не только в вашем классе, но и в надклассе тоже), которые должны как-то указывать, какие части массива использует ваш класс. В противном случае можно обратиться к уже занятой позиции.

Со всеми этими моментами помогает справиться прагма `use fields`:

```
package Person;
use fields qw(name race aliases);
```

Эта прагма не создает методы доступа вместо вас, но использует некоторые встроенные механизмы (называемые *псевдохешами*), выполняющие нечто подобное. (Во всяком случае можно создать методы доступа – оболочки для полей, как делается в следующем примере.) Псевдохеши – это ссылки на массивы, которые можно использовать как хеши, поскольку с ними связывается таблица отображения ключей. Это отображение ключей устанавливается с помощью прагмы `use fields`, которая объявляет, какие поля допустимы в объекте `Person`; благодаря этому о них знает компилятор Perl. Если объявить тип вашей переменной объекта (как в `my Person $self` в следующем примере), то компилятор оказывается достаточно сообразительным, чтобы оптимизировать доступ к полям, превратив его в прямой доступ к массиву. Более важно, вероятно, что он проверяет имена полей на правильность типа (в действительности на отсутствие опечаток при вводе) на этапе компиляции. (См. раздел «Псевдохеши» главы 8 «Ссылки»).

Конструктор и типовые методы доступа могут выглядеть так:

```
package Person;
use fields qw(name race aliases);
sub new {
    my $type = shift;
    my Person $self = fields::new(ref $type || $type);
    $self->{name} = "unnamed";
    $self->{race}  = "unknown";
    $self->{aliases} = [];
    return $self;
}
sub name {
    my Person $self = shift;
    $self->{name} = shift if @_;
    return $self->{name};
}
sub race {
    my Person $self = shift;
    $self->{race} = shift if @_;
    return $self->{race};
}
sub aliases {
    my Person $self = shift;
    $self->{aliases} = shift if @_;
    return $self->{aliases};
}
1;
```

Если неправильно введен литеральный ключ для доступа к псевдохешу, то не придется ждать этапа исполнения, чтобы узнать об этом. Компилятор знает тип объекта, на который должна ссылаться `$self` (вы сами это сообщили), поэтому может проверить, что код обращается только к тем полям, которые действительно есть у объектов `Person`. Если у вас ветер в голове и вы пытаетесь обратиться к несуществующему полю (например, `$self->{mane}`),

компилятор сразу сигнализирует об этой ошибке и не передает ошибочную программу на выполнение интерпретатору.

Методы получения доступа к переменным экземпляра несколько повторяются, поэтому целесообразно автоматизировать создание простых методов доступа с помощью предлагаемых ниже приемов. Однако, поскольку во всех этих методах используется того или иного рода косвенная адресация, в случае их применения теряются преимущества проверки опечаток в ключах доступа к хешу на этапе компиляции.

Если вы для реализации своего класса выбрали псевдохеш, то любой класс, наследующий ему, должен знать о том, что в основе реализации лежит псевдохеш. Если объект реализован как псевдохеш, то все участники иерархии наследования должны использовать объявления `use base` и `use fields`. Например,

```
package Wizard;
use base "Person";
use fields qw(staff color sphere);
```

Этот код делает модуль `Wizard` подклассом класса `Person` и загружает файл *Person.pm*. Кроме того, регистрируются также три новых поля в этом классе в дополнение к унаследованным от `Person`. Поэтому, когда вы пишете:

```
my Wizard $mage = fields::new("Wizard");
```

то получаете объект псевдохеша с доступом к полям обоих классов:

```
$mage->name("Gandalf");
$mage->color("Grey");
```

Поскольку всем подклассам должно быть известно, что они применяют реализацию в виде псевдохеша, они должны использовать непосредственную нотацию псевдохеша как для эффективности, так и для проверки ввода:

```
$mage->{name} = "Gandalf";
$mage->{color} = "Grey";
```

Однако если вы хотите, чтобы ваши реализации можно было изменять, сторонние пользователи вашего класса должны применять для доступа к полям методы.

Хотя `use base` поддерживает только одиночное наследование, это редко является серьезным ограничением. См. описания `use base` и `use fields` в главе 31 «Модули прагм».

Порождение классов с помощью Class::Struct

Стандартный модуль `Class::Struct` экспортирует функцию с именем `struct`. Она создает все необходимое для начала работы с классом в целом. Ею генерируется конструктор с именем `new`, а также методы доступа ко всем полям данных (переменным экземпляра), указанным в этой структуре.

Например, поместим класс в файл *Person.pm*:

```
package Person;
use Class::Struct;
struct Person => {    # создать определение для "Person"
    name    => '$',   #    поле name является скаляром
    race    => '$',   #    поле race тоже является скаляром
    aliases => '@',   #    но поле aliases является ссылкой на массив
};
1;
```

Теперь можно так использовать этот модуль:

```
use Person;
my $mage = Person->new();
$mage->name("Gandalf");
$mage->race("Istar");
$mage->aliases( ["Mithrandir", "Olorin", "Incanus"] );
```

Все эти четыре метода создал модуль `Class::Struct`. Поскольку он придерживается политики защиты подклассов путем дополнения имени поля префиксом из имени класса, он позволяет классу-наследнику, не вызывая конфликта, иметь свое собственное поле с таким же именем, как у базового класса. В данном случае это значит, что в качестве ключа хеша для переменной этого экземпляра используется `Person::name`, а не просто `name`.

Поля в объявлении `struct` не обязательно должны иметь базовый тип Perl. Они могут также указывать на другие классы, но лучше всего работают классы, созданные с помощью `struct`, потому что эта функция делает допущения о поведении классов, которые, вообще говоря, верны не для всех них. Например, для инициализации поля вызывается метод `new` соответствующего класса, хотя для многих классов конструкторы имеют другие имена.

Дополнительные сведения можно получить в описании `Class::Struct` в главе 32 «Стандартные модули» и электронной документации. Многие стандартные модули с помощью `Class::Struct` реализуют свои классы, в том числе `User::pwent` и `Net::hostent`. Изучение их кода может оказаться поучительным.

Генерация методов доступа с помощью автозагрузки

Как говорилось выше, при вызове несуществующего метода у Perl есть два способа поиска метода `AUTOLOAD` в зависимости от того, объявлен ли метод-заглушка. Это свойство можно использовать для предоставления доступа к данным экземпляра без написания отдельной функции для каждого экземпляра. В программе `AUTOLOAD` имя фактически вызываемого метода можно извлечь из переменной `$AUTOLOAD`. Рассмотрим следующий код:

```
use Person;
$him = Person->new;
$him->name("Aragorn");
$him->race("Man");
$him->aliases( ["Strider", "Estel", "Elessar"] );
printf "%s is of the race of %s.\n", $him->name, $him->race;
print "His aliases: ", join(", ", @{$him->aliases}), ".\n";
```

Как и прежде, эта версия класса Person реализует структуру данных с тремя полями: name, race и aliases:

```
package Person;
use Carp;

my %Fields = (
    "Person::name"  => "unnamed",
    "Person::race"   => "unknown",
    "Person::aliases"  => [],
);

# Следующее объявление гарантирует получение собственного автозагрузчика.
use subs qw(name race aliases);

sub new {
    my $invocant = shift;
    my $class = ref($invocant) || $invocant;
    my $self  = { %Fields, @_ };    # клонировать, как Class::Struct
    bless $self, $class;
    return $self;
}

sub AUTOLOAD {
    my $self = shift;
    # обрабатывать только методы экземпляра, но не класса
    croak "$self не является объектом" unless ref($invocant);
    my $name = our $AUTOLOAD;
    return if $name =~ /::DESTROY$/;
    unless (exists $self->{$name}) {
        croak "Нет доступа к полю `$name' в $self";
    }
    if (@_) { return $self->{$name} = shift }
    else    { return $self->{$name} }
}
```

Как видите, методов с именами name, race или aliases нигде нет. Обо всем этом заботится программа AUTOLOAD. Если кто-то использует $him->name("Aragorn"), подпрограмма AUTOLOAD вызывается с $AUTOLOAD, установленной в Person::name. То, что имя полностью квалифицировано, приводит его как раз в тот формат, который нужен для доступа к полям хеша объекта. Благодаря этому в случае применения данного класса в составе более крупной иерархии не возникает конфликта с такими же именами в других классах.

Создание методов доступа с помощью замыканий

Большинство методов доступа делают, в сущности, одно и то же: они просто выбирают значение из переменной экземпляра или записывают его туда. В Perl наиболее естественный способ создания семейства почти повторяющих одна другую функций состоит в цикле по замыканию. Но замыкания представляют собой анонимные функции, у которых нет имен, а методы должны быть именованными подпрограммами в таблице имен пакета класса, чтобы к ним можно было обращаться по имени. Эта проблема решается легко: нужно присвоить ссылку на замыкание переменной typeglob с нужным именем.

```
package Person;

sub new {
    my $invocant = shift;
    my $self = bless({}, ref $invocant || $invocant);
    $self->init();
    return $self;
}

sub init {
    my $self = shift;
    $self->name("unnamed");
    $self->race("unknown");
    $self->aliases([]);
}

for my $field (qw(name race aliases)) {
    my $slot = __PACKAGE__ . "::$field";
    no strict "refs";          # Так работает символическая ссылка на typeglob.
        my $self = shift;
        $self->{$field} = shift if @_;
        return $self->{$field};
    };
}
```

Замыкания служат самым компактным ручным способом создания группы методов доступа к данным экземпляра. Он эффективен как для компьютера, так и для программиста. Методы доступа не только совместно используют один и тот же код (им только нужна собственная лексическая память), но и требуют минимальных изменений, если впоследствии понадобится добавить новый атрибут: нужно лишь добавить еще одно слово к списку цикла и, возможно, что-нибудь к методу `init`.

Использование замыканий в закрытых объектах

Приведенные до сих пор приемы управления данными экземпляров не предоставляли механизма «защиты» от внешнего доступа. Находясь вне класса, каждый может открыть черный ящик объекта и залезть внутрь – если он не опасается потери гарантии. Осуществление политики закрытия может

привести к тому, что она станет поперек дороги тем, кто ищет, как решить стоящую перед ними задачу. Философия Perl состоит в том, что инкапсуляцию данных лучше производить с помощью надписи, которая гласит:

```
В СЛУЧАЕ ПОЖАРА
 РАЗБИТЬ СТЕКЛО
```

К такого рода инкапсуляции следует, по возможности, относиться с уважением, но, тем не менее, нужно иметь простой доступ к содержимому в аварийной ситуации – например, для отладки.

Но если вы действительно хотите обеспечить закрытость, Perl не собирается вам мешать. Perl предоставляет строительные блоки низкого уровня, которые можно использовать, чтобы окружить класс и его объекты непроницаемым щитом закрытости – в действительности, более прочным, чем тот, который обеспечивают многие распространенные объектно-ориентированные языки. Ключевыми компонентами здесь служат лексические области видимости и лексические переменные в них, а замыкания играют решающую роль.

В разделе «Закрытые методы» мы разобрали, как класс может использовать замыкания для реализации методов, которые не видны вне файла модуля. Ниже мы рассмотрим методы доступа, которые делают данные класса настолько закрытыми, что даже остальную часть класса лишают неограниченного доступа. Но все же это достаточно традиционное применение замыканий. Действительно интересный подход состоит в том, чтобы использовать замыкание в качестве самого объекта. Переменные экземпляра объекта оказываются заточенными в области видимости, к которой имеет свободный доступ только сам объект – т. е. замыкание. Это очень сильная форма инкапсуляции: она не только защищает от вмешательства извне, но и требует, чтобы другие методы того же класса получали данные объекта посредством надлежащих методов доступа.

Вот пример использования этой технологии. Мы применяем замыкания как для самих объектов, так и для создаваемых методов доступа:

```
package Person;
sub new {
    my $invocant  = shift;
    my $class = ref($invocant) || $invocant;
    my $data = {
       NAME     => "unnamed",
       RACE     => "unknown",
       ALIASES  => [],
    };
    my $self = sub {
       my $field = shift;
       ##############################
       ### ЗДЕСЬ ПРОВЕРКА ДОСТУПА ###
       ##############################
       if (@_) { $data->{$field} = shift }
       return    $data->{$field};
```

```
        };
        bless($self, $class);
        return $self;
    }
    # создание имен методов
    for my $field (qw(name race aliases)) {
        no strict "refs";  # для доступа к таблице имен    *$field = sub {
            my $self = shift;
            return $self->(uc $field, @_);
        };
    }
```

Объект, создаваемый и возвращаемый методом new, уже не является хешем, как это было в рассмотренных нами ранее конструкторах. Это замыкание с исключительным доступом к данным атрибутов, хранящимся в хеше, на который ссылается $data. После завершения вызова конструктора доступ к $data (и, следовательно, атрибутам) возможен только через замыкание.

При обращении вида $him->name("Bombadil") вызывающий объект, хранящийся в $self, является замыканием, обработанным bless и возвращенным конструктором. С замыканием мало что можно сделать, кроме как вызвать его; это мы и делаем с помощью $self->(uc $field, @_). Пусть вас не вводит в заблуждение стрелка: это обычное косвенное обращение к функции, а не вызов метода. Первым аргументом является строка name, а оставшиеся аргументы представляют все, что еще было передано.[1] Когда выполнение переходит внутрь замыкания, ссылка на хеш внутри $data снова становится доступной. После этого замыкание имеет возможность разрешать доступ или отказывать в нем по своему усмотрению.

Никто за пределами объекта замыкания не имеет непосредственного доступа к этим сильно закрытым данным экземпляра, даже другие методы класса. Они могли бы попытаться обратиться к замыканию таким же способом, как это делают методы, генерируемые в цикле for, например, присваивая значение переменной экземпляра, о которой классу ничего не известно. Но такой подход легко блокируется в участках кода конструктора, где можно найти комментарии по поводу проверки доступа. Прежде всего, нужна общая преамбула:

```
use Carp;
local $Carp::CarpLevel = 1;  # Делает сообщения croak короткими
my ($cpack, $cfile) = caller();
```

Теперь рассмотрим каждую из этих проверок. Первая проверяет существование указанного имени атрибута:

```
croak "Недопустимое поле '$field' в объекте"
    unless exists $data->{$field};
```

[1] Конечно, двойное обращение к функциям выполняется медленно, но если вам важна скорость, станете ли вы вообще использовать объекты?

Следующая проверка разрешает вызов только из того же файла:

```
carp "Непосредственный доступ из сторонних файлов запрещен"
    unless $cfile eq __FILE__;
```

Следующая проверка разрешает вызов только из того же пакета:

```
carp "Непосредственный доступ запрещен стороннему пакету ${cpack}::"
    unless $cpack eq __PACKAGE__;
```

А эта проверка разрешает вызов только из классов, наследующих нашему:

```
carp "Непосредственный доступ запрещен недружественному классу ${cpack}::"
    unless $cpack->isa(__PACKAGE__);
```

Все эти проверки блокируют только доступ без посредника. Пользователи класса, которые вежливо применяют методы, предназначенные для доступа, таких ограничений не имеют. Perl предоставляет вам средства проявить свою привередливость в той мере, как вы того хотите. К счастью, хотят не столь многие.

Но некоторым приходится быть разборчивыми. Разборчивость нужна тому, кто пишет программное обеспечение для управления полетами. Если вы хотите или обязаны быть одним из таких людей и предпочитаете использовать готовый код, а не изобретать все самому, возьмите из CPAN модуль `Tie::SecureHash` Дамиана Конвея (Damian Conway). В нем реализованы хеши с ограниченным доступом, поддерживающие разборчивость на открытом, защищенном и закрытом уровнях. В нем решаются также проблемы наследования, которые мы проигнорировали в предыдущем примере. Дамиан написал также еще более амбициозный модуль `Class::Contract`, который налагает на гибкую систему объектов Perl формальный режим разработки программного обеспечения. Список функций этого модуля выглядит, как перечень из учебника по разработке программного обеспечения, написанного профессором информатики,[1] включая принудительную инкапсуляцию, статическое наследование и проверку условия проектирования по контракту (design-by-contract condition) для объектно-ориентированного Perl наряду с декларативным синтаксисом для определений атрибутов, методов, конструкторов и деструкторов на уровне объекта и класса, а также предусловиями, постусловиями и инвариантами классов. Уф!

Новые приемы

Начиная с версии Perl 5.6, можно объявлять методы, возвращающие l-значение. Это делается с помощью атрибута подпрограммы lvalue (не путайте с атрибутами объекта). Эта экспериментальная функция позволяет обращаться с методом как с чем-то, что может появиться слева от знака равенства:

---

[1] Попробуйте угадать, кем работает Дамиан. Между прочим, весьма рекомендуем его книгу «Object Oriented Perl» (Manning Publications, 1999).

```
package Critter;

sub new {
    my $class = shift;
    my $self = { pups => 0, @_ };     # Замещение значения по умолчанию.
    bless $self, $class;
}

sub pups : lvalue {                   # Присвоение pups() будет выполнено позже.
    my $self = shift;
    $self->{pups};
}

package main;
$varmint = Critter->new(pups => 4);
$varmint->pups *= 2;                  # Присвоение $varmint->pups!
$varmint->pups =~ s/(.)/$1$1/;        # Изменение $varmint->pups по месту!
print $varmint->pups;                 # Теперь у нас 88 pups.
```

Это позволяет обращаться с `$varmint->pups` как с переменной, в то же время придерживаясь инкапсуляции. См. раздел «Атрибут lvalue» главы 6 «Подпрограммы».

Если вы работаете с версией Perl, поддерживающей потоки, и хотите, чтобы только один поток мог вызывать некоторый метод с объектом, можете использовать для этого атрибуты `locked` и `method`:

```
sub pups : locked method {
    ...
}
```

Когда поток вызывает с объектом метод `pups`, Perl перед его выполнением блокирует объект, запрещая делать это другим объектам. См. раздел «Атрибуты locked и method» главы 6 «Подпрограммы».

Управление данными класса

Мы рассмотрели несколько подходов для доступа к данным в объектах. Однако иногда нужно, чтобы существовало некоторое состояние, общее для всех объектов класса. Такие переменные являются глобальными для целого класса, а не атрибутами лишь одного его экземпляра, и не зависят от того, через какой экземпляр класса (объект) осуществляется к ним доступ. (Программисты C++ могут представить их себе, как статические члены-данные.) Вот некоторые ситуации, в которых может оказаться удобным использование переменных класса:

- Ведение учета всех когда-либо созданных или действующих в данный момент объектов.
- Ведение списка всех объектов, которые можно итерировать.
- Хранение имени дескриптора файла для журнального файла, используемого методом отладки всего класса.

- Ведение учета общих данных, например общей суммы денег, выданных всеми банкоматами в сети за данные сутки.
- Отслеживание последнего созданного классом объекта или объекта, к которому чаще всего осуществляется доступ.
- Поддержка кэша хранимых в памяти объектов, воссозданных из постоянной памяти.
- Поддержка таблицы обратного поиска для обнаружения объекта по значению его атрибута.

Вопрос сводится к принятию решения о том, где хранить состояние этих общих атрибутов. В Perl нет специального синтаксического механизма для объявления как атрибутов класса, так и атрибутов экземпляра. Perl предоставляет разработчику набор мощных, но гибких функций, которые можно уникальным образом использовать в зависимости от конкретных требований ситуации. В результате можно выбрать механизм, наиболее оправданный в данной ситуации, а не полагаться на чужие проектные решения. С другой стороны, можно обратиться к проектным решениям, которые кто-то другой упаковал и положил в CPAN. Как всегда, TMTOWTDI.

Как и во всем, что относится к классам, при доступе к данным класса следует избегать непосредственного обращения к ним, особенно из-за пределов самой реализации класса. Нельзя обеспечить инкапсуляцию, если, тщательно разработав ограничивающие методы доступа к переменным экземпляра, затем позволить кому угодно укокошить свои переменные класса непосредственно, например, установив `$SomeClass::Debug = 1`. Чтобы установить четкую преграду между интерфейсом и реализацией, можно создать методы доступа к данным класса, аналогичные тем, которые применяются для доступа к данным экземпляра.

Представим себе, что нужно вести глобальный учет поголовья объектов `Critter`. Мы будем хранить это число в переменной пакета, но предоставим метод с именем `population`, благодаря которому пользователям класса не потребуется знать ничего о механизме реализации.

```
Critter->population()       # Доступ через имя класса
$gollum->population()       # Доступ через экземпляр
```

Поскольку класс в Perl является просто пакетом, естественнее всего хранить данные класса в переменной пакета. Ниже приведена простая реализация такого класса. Метод `population` игнорирует передаваемый ему инвокант и возвращает текущее значение переменной пакета, `$Population`. (Некоторые программисты предпочитают писать свои глобальные переменные с заглавной буквы.)

```
package Critter;
our $Population = 0;
sub population { return $Population; }
sub DESTROY { $Population- - }
sub spawn {
```

```
    my $invocant = shift;
    my $class = ref($invocant) || $invocant;
    $Population++;
    return bless { name => shift || "anon" }, $class;
}
sub name {
    my $self = shift;
    $self->{name} = shift if @_;
    return $self->{name};
}
```

Для того чтобы методы данных класса работали как методы доступа к данным, следует делать так:

```
our $Debugging = 0;     # начальные данные класса
sub debug {
    shift;              # намеренное игнорирование инвоканта
    $Debugging = shift if @_;
    return $Debugging;
}
```

Теперь можно установить общий уровень отладки в классе или любом из его экземпляров.

Являясь переменной пакета, `$Debugging` доступна глобально. Но если изменить `our` на `my`, то ее сможет видеть только последующий код в том же файле. Можно пойти еще дальше: отобрать неограниченный доступ к атрибутам класса даже у остальной части самого класса. Поместите объявление переменной в область видимости блока:

```
{
    my $Debugging = 0;          # начальные данные класса с лексической областью
видимости
    sub debug {
        shift;                  # намеренное игнорирование инвоканта
        $Debugging = shift if @_;
    }
}
```

Теперь никто не сможет читать или записывать атрибуты класса без использования метода доступа, т. к. только его подпрограмма находится в той же области видимости, что и переменная, и имеет к ней доступ.

Если производный класс наследует эти методы доступа к данным класса, то они осуществляют доступ к исходным данным независимо от того, определялись ли переменные с помощью `our` или `my`. Данные не зависят от пакета. Можно рассматривать это как выполнение методов в классе, в котором они были первоначально определены, а не в классе, вызвавшем их.

Для одних типов классов такой подход работает, а для других – нет. Допустим, мы создаем подкласс `Warg` класса `Critter`. Если мы хотим вести отдельный учет поголовья, то `Warg` не может наследовать метод population у `Critter`,

поскольку этот метод, как мы его написали, всегда возвращает значение `$Critter::Population`.

Необходимо решать в каждом конкретном случае отдельно, имеет ли смысл атрибутам класса зависеть от пакета. Если вам нужны атрибуты, зависящие от пакета, используйте класс инвоканта для нахождения пакета с данными класса:

```
sub debug {
    my $invocant = shift;
    my $class   = ref($invocant) || $invocant;
    my $varname = $class . "::Debugging";
    no strict "refs";       # для доступа к данным пакета через таблицу имен
    $$varname = shift if @_;
    return $$varname;
}
```

Мы временно отменяем ограничение на ссылки, потому что иначе не смогли бы использовать полностью квалифицированное символическое имя для глобальной переменной пакета. Это совершенно обоснованно: поскольку, по определению, все пакетные переменные располагаются в пакете, нет ничего плохого в обращении к ним через таблицу имен этого пакета.

Другой подход состоит в том, чтобы сделать все необходимые объекту данные – даже глобальные данные класса – доступными через этот объект (или передавать в качестве параметров). С этой целью часто приходится создавать специальный конструктор для каждого класса или, по крайней мере, специальную процедуру инициализации, вызываемую конструктором. Конструктор или процедура инициализации записывают ссылки на любые данные класса непосредственно в сам объект, поэтому в будущем их не нужно будет искать. Методы доступа берут ссылки на данные из объекта.

Каждый метод не ведет сложный поиск данных класса; вместо этого объект сообщает методу, где эти данные находятся. Этот метод хорошо работает, только если методы доступа к данным класса вызываются как методы экземпляра, потому что данные класса могут храниться в недоступных лексических переменных, до которых нельзя добраться с помощью имени пакета.

Как ни крути, а данные класса, зависящие от пакета, всегда несколько опасны. Значительно более надежно, если при наследовании метода доступа к данным класса наследуются и данные состояния, к которым он обращается. Многочисленные и более проработанные подходы к управлению данными класса можно найти на странице руководства *perltootc*.

Резюме

Это почти все, что касается данной темы, за исключением всего прочего. Вам остается лишь пойти и купить книгу по методам объектно-ориентированного проектирования и биться об нее лбом ближайшие полгода или около того.

13

Перегрузка

Объекты, конечно, хороши, но иногда они *слишком* хороши. Иногда хочется, чтобы они меньше вели себя как объекты и больше как обычные типы данных. Но на этом пути есть проблема: объекты представляются ссылками, а возможности работы со ссылками не велики: ссылки нельзя складывать, выводить или (с какой-нибудь пользой) применять многие встроенные операторы Perl. По сути дела, их можно только разыменовывать. Вот и приходится то и дело явно писать вызовы методов, например:

```
print $object->as_string;
$new_object = $subject->add($object);
```

Такое явное разыменование, вообще говоря, дело полезное, т. к. при этом не спутаешь ссылки с их объектами, кроме как в тех случаях, когда это делается намеренно. Один из таких моментов сейчас и настал. Если при разработке класса использована *перегрузка* (*overloading*), то можно сделать вид, что никаких ссылок нет, и просто сказать:

```
print $object;
$new_object = $subject + $object;
```

При перегрузке встроенного оператора Perl вы определяете, как он должен себя вести, будучи применен к объектам определенного класса. Перегрузка применяется в ряде стандартных модулей Perl, например `Math::BigInt`, который позволяет создавать объекты `Math::BigInt`, ведущие себя как обычные целые числа, но размер которых не ограничен. Их можно складывать

с помощью +, делить посредством /, для сравнения использовать <=>, а для вывода – print.

Обратите внимание, что перегрузка (overloading) – это совсем не то же самое, что автозагрузка (autoloading), которая означает загрузку отсутствующих функций или методов по требованию. Точно так же, это совсем не то же самое, что замещение (overriding), при котором одна функция или метод маскируют другую функцию. Перегрузка ничего не скрывает; благодаря ей бессмысленная до того операция над ссылкой приобретает смысл.

Прагма overload

Прагма use overload осуществляет перегрузку операторов. Ей передается список пар ключ/значение, состоящий из операторов и связанных с ними действий:

```
package MyClass;

use overload   '+' => \&myadd,            # ссылка на код
               '<' => "less_than",        # именованный метод
            'abs' => sub { return @_ };   # анонимная подпрограмма
```

Теперь, если попытаться сложить два объекта MyClass, то для формирования результата будет вызвана подпрограмма myadd.

Если вы попытаетесь сравнить два объекта MyClass с помощью оператора <, то Perl обнаружит, что нужное действие задано в виде строки и будет интерпретировать эту строку как имя метода, а не просто подпрограммы. В приведенном примере метод less_than может быть предоставлен самим пакетом MyClass или унаследован от базового для MyClass класса, а вот подпрограмма myadd должна быть предоставлена текущим пакетом. Анонимная подпрограмма для abs предоставляется еще более непосредственным способом. Каким бы образом ни предоставлялись эти программы, будем называть их *обработчиками* (*handlers*).

Для унарных операторов (т. е. принимающих один аргумент, как abs) указанный для класса обработчик вызывается всякий раз, когда этот оператор применяется к объекту этого класса.

Для бинарных операторов, таких как + или <, обработчик вызывается, если первый операнд является объектом класса *или* второй операнд является объектом класса, а для первого операнда перегрузка не задана. Поэтому можно сказать:

```
$object + 6
```

или:

```
6 + $object
```

не беспокоясь о порядке операндов. (Во втором случае операнды при передаче обработчику *поменяются местами*.) Если бы у нас было выражение:

```
$animal + $vegetable
```

и `$animal` (животное) и `$vegetable` (овощ) были бы объектами разных классов, в обоих из которых используется перегрузка, то в действие была бы введена перегрузка `$animal`. (Будем надеяться, что животному овощи понравятся.)

В Perl есть только один тернарный («тринарный») оператор, `?:`, и перегрузить его нельзя. К счастью.

Обработчики перегрузки

При выполнении перегружаемого оператора соответствующий обработчик вызывается с передачей ему трех аргументов. Первые два аргумента суть операнды. Если оператор использует только один аргумент, то вторым аргументом является `undef`.

Третий аргумент указывает на то, могут ли быть переставлены первые два аргумента. Даже по правилам обычной арифметики некоторые операции, например сложение и умножение, безразличны к порядку своих аргументов, а другие, например вычитание и деление, нет.[1] Взгляните на разницу между:

```
$object - 6
```

и:

```
6 - $object
```

Если первые два аргумента обработчика могут меняться местами, то третий имеет значение true. В противном случае третий аргумент будет равен false, и в этом случае есть также более тонкое различие: если обработчик запущен другим обработчиком оператора с присваиванием (как для `+=`, где положение знака `+` определяет тип инкремента), то третий аргумент не просто false, но `undef`. Это различие позволяет выполнить некоторую оптимизацию.

В качестве примера приведем класс, позволяющий работать с ограниченным диапазоном чисел. В нем `+` и `-` перегружаются таким образом, что результат сложения или вычитания ограничивается, чтобы попасть в диапазон от 0 до 255:

```
package ClipByte;

use overload '+' => \&clip_add,
```

[1] Перегружаемые вами объекты не обязаны, конечно, придерживаться правил обычной арифметики, но вообще лучше не вызывать у людей удивление. Странно, что во многих языках совершают ошибку, перегружая `+` на конкатенацию строк, что не только не коммутативно, но даже аддитивно весьма отдаленно. В Perl вы найдете другой подход.

```
            '-' => \&clip_sub;

sub new {
    my $class = shift;
    my $value = shift;
    return bless \$value => $class;
}

sub clip_add {
    my ($x, $y) = @_;
    my ($value) = ref($x) ? $$x : $x;
    $value     += ref($y) ? $$y : $y;
    $value = 255 if $value > 255;
    $value =   0 if $value < 0;
    return bless \$value => ref($x);
}

sub clip_sub {
    my ($x, $y, $swap) = @_;
    my ($value) = (ref $x) ? $$x : $x;
    $value     -= (ref $y) ? $$y : $y;
    if ($swap) { $value = -$value }
    $value = 255 if $value > 255;
    $value =   0 if $value < 0;
    return bless \$value => ref($x);
}

package main;

$byte1 = ClipByte->new(200);
$byte2 = ClipByte->new(100);

$byte3 = $byte1 + $byte2;    # 255
$byte4 = $byte1 - $byte2;    # 100
$byte5 = 150 - $byte2;       # 50
```

Обратите внимание на то, что все функции здесь по необходимости являются конструкторами, поэтому каждая из них использует `bless` для связи ссылки на свой новый объект с текущим классом, чем бы он ни был; мы предполагаем, что наш новый класс может наследоваться. Мы также предполагаем, что если `$y` является ссылкой, то это ссылка на объект созданного нами типа. Если быть дотошнее, то вместо проверки `ref($y)` можно вызывать `$y->isa("ClipByte")` (выполнялось бы это медленнее).

Перегружаемые операторы

Перегружать можно только определенные операторы, которые приведены в табл. 13.1. Эти операторы перечислены также в хеше `%overload::ops`, который становится доступным в результате `use overload`, хотя разбивка по категориям производится там несколько иначе.

Таблица 13.1. Перегружаемые операторы

Категория	Операторы
Преобразование	`"" 0+ bool`
Арифметические	`+ - * / % ** x . neg`
Логические	`!`
Поразрядные	`& | ~ ^ ! << >>`
Присваивания	`+= -= *= /= %= **= x= .= <<= >>= ++ - -`
Сравнения	`== < <= > >= != <=> lt le gt ge eq ne cmp`
Математические	`atan2 cos sin exp abs log sqrt`
Итеративные	`<>`
Разыменования	`${} @{} %{} &{} *{}`
Псевдо	`nomethod fallback =>`

Обратите внимание, что `neg`, `bool`, `nomethod` и `fallback` фактически не являются операторами Perl. Пять разыменовывающих операторов, `""` и `0+` тоже *не похожи* на операторы. Тем не менее, все они допустимы в качестве ключей в списке параметров, передаваемом `use overload`. На самом деле, это не представляет проблемы. Откроем маленький секрет: мы слегка мошенничаем, говоря, что прагма `overload` перегружает операторы. Она перегружает производимые операции, когда вызывается явно через «официальные» операторы или неявно через какой-либо связанный оператор. (Отмеченные псевдооператоры могут вызываться только неявно.) Иными словами, перегрузка производится не на синтаксическом уровне, а на семантическом. Цель не в том, чтобы хорошо выглядеть, а в том, чтобы делать то, что нужно. Обобщить можете сами.

Заметьте также, что `=` *не* перегружает оператор присваивания Perl, как можно было бы предположить. Это не было бы Правильным Делом. Подробнее об этом ниже.

Мы начнем с обсуждения операторов преобразования — не потому, что их действие очевиднее всего (это не так), а потому что они самые полезные. Многие классы перегружают только преобразование в строку, задаваемое ключом `""`, и больше ничего. (Да, это действительно две двойных кавычки подряд.)

Операторы преобразования: `""`, `0+`, `bool`.

> Эти три ключа позволяют задать для Perl режимы автоматического преобразования в строки, числа и булевы значения, соответственно.
>
> Мы говорим, что происходит *преобразование в строку* (*stringification*), если нестроковая переменная используется в качестве строки. Подобные вещи имеют место в случае преобразования переменной в строку при выводе, интерполяции, конкатенации и даже при использовании в качестве

ключа хеша. Преобразование также служит причиной того, что вы видите что-то типа SCALAR(0xba5fe0) при попытке вывести объект.

Мы говорим, что происходит *преобразование в число* (*numification*), когда нечисловая переменная преобразуется в число в каком-либо числовом контексте, например, в математическом выражении, индексе массива или даже операторе .. диапазона.

Наконец, хотя никто не посмеет назвать это *boolification*, но можно определить способ интерпретации объекта в булевом контексте (таком, как if, unless, while, for, and, or, &&, ||, ?: или блок выражения grep) путем создания булева обработчика.

Любые из этих трех операторов преобразования могут *самогенерироваться*, если имеется один из них (что такое самогенерация, мы разъясним позже). Ваши обработчики могут возвращать любые значения. Отметьте, что если операция, запустившая преобразование, тоже перегружена, то ее перегрузка происходит сразу после преобразования.

Вот пример "", который запускает обработчик as_string объекта при преобразовании в строку. Не забывайте заключать в кавычки сами кавычки:

```
package Person;
use overload q("") => \&as_string;
sub new {
    my $class = shift;
    return bless { @_ } => $class;
}

sub as_string {
    my $self = shift;
    my ($key, $value, $result);
    while (($key, $value) = each %$self) {
        $result .= "$key => $value\n";
    }
    return $result;
}

$obj = Person->new(height => 72, weight => 165, eyes => "темно-карие");

print $obj;
```

Вместо чего-то вроде Person=HASH(0xba1350) этот код выводит (в порядке, определяемом хешем):

```
weight => 165
height => 72
eyes => темно-карие
```

(Искренне надеемся, что вес и рост этого человека выражаются не в килограммах и сантиметрах.)

Арифметические операторы: +, -, *, /, %, **, x, ., neg

Все они должны быть вам знакомы, за исключением neg, который является специальным ключом перегрузки унарного минуса: знак - в -123. Раз-

личие между ключами neg и - позволяет задать различный режим для унарного минуса и бинарного минуса, обычно называемого вычитанием.

Если вы перегружаете -, но не neg, а затем пытаетесь использовать унарный минус, то Perl эмулирует для вас обработчик neg. Это называется *самогенерацией* (*autogeneration*); при этом некоторые операторы можно разумным образом вывести из других (в допущении, что перегруженные операторы будут иметь такие же связи, как у обычных операторов.) Так как унарный минус можно выразить как функцию бинарного минуса (то есть -123 эквивалентно 0 - 123), то Perl не заставляет вас перегружать neg, поскольку можно обойтись -. (Конечно, если вы определили, что бинарный минус делит второй аргумент на первый, то унарный минус служит отличным способом возбуждения исключительной ситуации деления на ноль.)

Конкатенация через оператор . может самогенерироваться через обработчик преобразования в строку (смотри "" выше).

Логический оператор: !

Если обработчик для ! не задан, он может самогенерироваться с помощью обработчиков bool, "" или 0+. Если вы перегружаете оператор !, оператор not тоже будет запускать определенный вами режим. (Помните наш маленький секрет?)

Вас может удивить отсутствие других логических операторов, но большинство логических операторов не могут перегружаться, потому что они имеют короткое действие. На самом деле они являются операторами управления порядком выполнения инструкций, которые должны иметь возможность откладывать выполнение некоторых своих аргументов. Это также служит причиной того, что оператор ?: не перегружается.

Поразрядные операторы: &, |, ~, ^, <<, >>

Оператор ~ является унарным, а все остальные – бинарными. Вот как можно перегрузить >> для выполнения чего-то вроде chop:

```
package ShiftString;

use overload
    '>>' => \&right_shift,
    '""' => sub { ${ $_[0] } };

sub new {
    my $class = shift;
    my $value = shift;
    return bless \$value => $class;
}

sub right_shift {
    my ($x, $y) = @_;
    my $value = $$x;
    substr($value, -$y) = "";
    return bless \$value => ref($x);
}
```

```
$camel = ShiftString->new("Camel");
$ram = $camel >> 2;
print $ram;            # Cam
</para>
```

Операторы присваивания: `+=`, `-=`, `*=`, `/=`, `%=`, `**=`, `x=`, `.=`, `<<=`, `>>=`, `++`, `--`

Эти операторы присваивания могут изменять значение своих аргументов или оставлять их как есть. Результат присваивается левому операнду, только если новое значение отличается от старого. Благодаря этому один и тот же обработчик может использоваться для перегрузки как `+=`, так и `+/`. Это допустимо, но редко рекомендуется, поскольку согласно семантике, описываемой далее в разделе «Когда отсутствует обработчик перегрузки (nomethod и fallback)», Perl в любом случае вызовет обработчик для `+`, предполагая что `+=` непосредственно не перегружен.

Конкатенация (`.=`) может самогенерироваться с использованием преобразования в строку, за которым следует обычная конкатенация строк. Операторы `++` и `--` могут самогенерироваться из `+` и `-` (или `+=` и `-=`).

Предполагается, что аргументы в обработчиках `++` и `--` *мутируют* (изменяются). В обработчике можно сделать так, чтобы автодекрементирование работало как с буквами, так и с числами:

```
package MagicDec;

use overload
    q(- -) => \&decrement,
    q("") => sub { ${ $_[0] } };

sub new {
    my $class = shift;
    my $value = shift;
    bless \$value => $class;
}

sub decrement {
    my @string = reverse split(//, ${ $_[0] } );
    my $i;
    for ($i = 0; $i < @string; $i++ ) {
        last unless $string[$i] =~ /a/i;
        $string[$i] = chr( ord($string[$i]) + 25 );
    }
    $string[$i] = chr( ord($string[$i]) - 1 );
    my $result = join('', reverse @string);
    $_[0] = bless \$result => ref($_[0]);
}

package main;

for $normal (qw/perl NZ Pa/) {
    $magic = MagicDec->new($normal);
    $magic- -;
    print "$normal стал $magic\n";
}
```

Этот код выводит:

```
perl стал perk
NZ стал NY
Pa стал Oz
```

в точности обратно волшебному оператору Perl автоинкрементирования строки.

Операция `++$a` может самогенерироваться с использованием `$a += 1` или `$a = $a + 1`, а `$a--` — с использованием `$a -= 1` или `$a = $a - 1`. Однако при этом не запускается режим копирования, который должен осуществлять настоящий оператор `++`. См. раздел «Конструктор копирования» далее в этой главе.

Операторы сравнения: `==`, `<`, `<=`, `>`, `>=`, `!=`, `<=>`, `lt`, `le`, `gt`, `ge`, `eq`, `ne`, `cmp`

Если оператор `<=>` перегружен, он может применяться для самогенерации режимов работы для `<`, `<=`, `>`, `>=`, `==` и `!=`. Аналогично, если перегружается `cmp`, он может использоваться для самогенерации режимов работы для `lt`, `le`, `gt`, `ge`, `eq` и `ne`.

Обратите внимание, что перегрузка `cmp` не позволяет так легко, как хотелось бы, сортировать объекты, поскольку сравниваться будут не сами объекты, а их строковые версии . Для того чтобы это сделать, потребуется также перегрузить `""` .

Математические функции: `atan2`, `cos`, `sin`, `exp`, `abs`, `log`, `sqrt`

Если функция `abs` недоступна, она может самогенерироваться из `<` или `<=>` в сочетании с унарным минусом или вычитанием.

Перегруженный `-` может применяться для самогенерации отсутствующих обработчиков для унарного минуса или `abs`, которая может тоже отдельно перегружаться. (Да, мы знаем, что `abs` напоминает функцию, а унарный минус напоминает оператор, но в Perl они не настолько различаются.)

Итеративный оператор: `<>`

Обработчик `<>` может запускаться путем использования `readline` (при чтении из дескриптора файла, как в `while (<FH>)`) или `glob` (при поиске файлов по шаблону, как в `@files = <*.*>`).

```
package LuckyDraw;

use overload
    '<>' => sub {
        my $self = shift;
        return splice @$self, rand @$self, 1;
    };

sub new {
    my $class = shift;
    return bless [@_] => $class;
}
```

```
package main;

$lotto = new LuckyDraw 1 .. 51;

for (qw(1-й 2-й 3-й 4-й 5-й 6-й)) {
    $lucky_number = <$lotto>;
    print "$_ счастливый номер: $lucky_number.\n";
}

$lucky_number = <$lotto>;
print "\nИ дополнительный номер: $lucky_number.\n";
```

В Калифорнии[1] этот код выводит:

```
1-й счастливый номер: 18
2-й счастливый номер: 11
3-й счастливый номер: 40
5-й счастливый номер: 51
6-й счастливый номер: 33
И дополнительный номер: 5
```

Операторы разыменования: `${}`, `@{}`, `%{}`, `&{}`, `*{}`

Попытки разыменования ссылок на скаляры, массивы, хеши, подпрограммы и glob могут перехватываться путем перегрузки этих пяти символов.

В электронной документации Perl по `overload` показано, как с помощью этого оператора моделировать собственные псевдохеши. Вот более простой пример, реализующий объект как анонимный массив, разрешающий ссылку через хеш. Не пытайтесь обращаться с ним как с настоящим хешем; вы не сможете удалять из объекта пары ключ/значение посредством `delete`. Для того чтобы соединить обозначения массива и хеша, используйте настоящий псевдохеш.

```
package PsychoHash;

use overload '%{}' => \&as_hash;

sub as_hash {
    my ($x) = shift;
    return { @$x };
}

sub new {
    my $class = shift;
    return bless [ @_ ] => $class;
}

$critter = new PsychoHash( height => 72, weight => 365, type => "camel" );

print $critter->{weight};   # prints 365
```

См. также главу 14 «Связанные переменные» о механизме, позволяющем переопределять базовые операции над хешами, массивами и скалярами.

---

[1] Речь идет о лотерее CALIFORNIA SUPERLOTTO. – *Примеч. науч. ред.*

При перегрузке оператора постарайтесь не создавать объекты со ссылками на самих себя. Например:

```
use overload '+' => sub { bless [ \$_[0], \$_[1] ] };
```

Это провоцирует неприятности, поскольку если сказать `$animal += $vegetable`, то в результате `$animal` станет ссылкой на превращенный в объект массив, первым элементом которого является `$animal`. Это пример *циклической ссылки* (*circular reference*), которая ведет к тому, что даже при уничтожении `$animal` занимаемая ею память не будет освобождена, пока не завершит работу ваш процесс (или интерпретатор). См. раздел «Сборка мусора, циклические и слабые ссылки» в главе 8 «Ссылки».

Конструктор копий (=)

Несмотря на внешний вид обычного оператора, `=` имеет особое и не вполне интуитивно понятное значение ключа перегрузки. Он *не* перегружает оператор присваивания Perl. Он не может это делать, потому что оператор присваивания зарезервирован для присваивания ссылкам, иначе все рухнет.

Обработчик для `=` используется в ситуациях, в которых мутатор (mutator) (например, `++`, `--` или любой из операторов присваивания) применяется к ссылке, использующей свой объект совместно с другой ссылкой. Обработчик `=` позволяет программисту перехватить мутатор и самому скопировать объект так, чтобы была изменена только копия. В противном случае будет разрушен оригинал.

```
$copy = $original;   # копирует только ссылку
++$copy;             # изменяет объект ссылки
```

Теперь терпите вместе с нами. Предположим, что `$original` представляет собой ссылку на объект. Чтобы заставить `++$copy` модифицировать только `$copy`, а не `$original`, сначала делается копия `$copy` и переменной `$copy` присваивается ссылка на этот новый объект. Эта операция осуществляется, только когда выполняется `++$copy`, поэтому `$copy` совпадает с `$original` до инкрементирования – но не после него. Иными словами, это `++`, который распознает необходимость копирования и обращается к конструктору копий.

Необходимость копирования распознают только мутаторы, такие как `++` или `+=`, либо `nomethod`, который описывается ниже. Если операция самогенерируется посредством `+`, как в:

```
$copy = $original;
$copy = $copy + 1;
```

то копирование не производится, потому что `+` не знает, что используется как мутатор.

Если во время выполнения некоторого мутатора требуется конструктор копии, но обработчик для `=` не задан, он может самогенерироваться как копи-

рование строки, при условии что объект является обычным скаляром, а не чем-то более замысловатым.

Например, код, фактически выполняемый для последовательности:

```
$copy = $original;
...
++$copy;
```

может в итоге привести к чему-то такому:

```
$copy = $original;
...
$copy = $copy->clone(undef, "");
$copy->incr(undef, "");
```

При этом предполагается, что `$original` указывает на перегруженный объект, ++ был перегружен на `\&incr`, а = перегружен на `\&clone`.

Аналогичные действия запускаются при выполнении `$copy = $original++`, что интерпретируется как `$copy = $original; ++$original`.

Когда обработчик перегрузки отсутствует (nomethod и fallback)

Когда к объекту применяется не перегруженный оператор, Perl сначала пытается самогенерировать его режим из других перегруженных операторов с помощью описанных выше правил. Если это не удается, Perl ищет режим перегрузки для `nomethod`, и если он существует, то использует его. Этот обработчик является для операторов тем же, чем подпрограмма `AUTOLOAD` для подпрограмм: это то, что вы делаете, когда ничего другого придумать не можете.

В случае применения `nomethod` за ним должна следовать ссылка на обработчик, принимающий четыре аргумента (а не три, как для всех остальных обработчиков). Первые три аргумента такие же, как для любого другого обработчика; четвертым является строка, соответствующая оператору, обработчик которого отсутствует. Он служит той же цели, что и переменная `$AUTOLOAD` в подпрограммах `AUTOLOAD`.

Если Perl вынужден искать обработчик для `nomethod`, но не находит его, то возбуждается исключительная ситуация.

Для того чтобы избежать самогенерации или сделать так, чтобы при неудачной попытке самогенерации перегрузка вообще не производилась, можно задать специальный ключ перегрузки `fallback` (перехода в аварийный режим). Используются три его состояния:

`undef`

Если `fallback` не установлен или явно установлен в `undef`, то последовательность событий при перегрузке остается неизменной: выполняется поиск обработчиков, делается попытка самогенерации и, наконец, вызыва-

ется обработчик nomethod. Если последнее оказывается неудачным, возбуждается исключительная ситуация.

false

Если значение fallback определено, но соответствует значению false (например, 0), самогенерация никогда не происходит. Perl вызовет обработчик nomethod, если он существует, и возбудит исключительную ситуацию в противном случае.

true

Это устанавливает почти такой же режим, как при undef, но при невозможности синтезировать надлежащий обработчик с помощью самогенерации исключительная ситуация не возбуждается. Вместо этого Perl возвращается к применению не перегруженного режима работы этого оператора, как если бы прагмы use overload в этом классе вообще не было.

Константы перегрузки

Можно изменить способ, которым Perl интерпретирует константы, с помощью overload::constant, что удобнее всего поместить в метод import пакета. (При этом следует надлежащим образом вызвать overload::remove_constant в методе unimport пакета, чтобы последний мог убрать после себя, когда вы его об этом попросите.)

Как overload::constant, так и overload::remove_constant предполагают передачу им списка пар ключ/значение. Ключами могут быть integer, float, binary, q и qr, а значениями должны быть имя подпрограммы, анонимная подпрограмма или ссылка на код, которые будут обрабатывать константы.

```
sub import { overload::constant ( integer => \&integer_handler,
                                  float   => \&float_handler,
                                  binary  => \&base_handler,
                                  q       => \&string_handler,
                                  qr      => \&regex_handler ) }
```

Обработчики, задаваемые для integer и float, будут вызываться, когда лексический анализатор Perl обнаружит число-константу. Это происходит независимо от прагмы use constant; простые команды типа

```
$year = cube(12) + 1;         # целое
$pi   = 3.14159265358979;     # плавающее
```

будут запускать те обработчики, которые вы указали.

Ключ binary позволяет перехватывать двоичные, восьмеричные и шестнадцатеричные константы. q обрабатывает строки в одинарных кавычках (в том числе строки, введенные с помощью q) и подстроки-константы в строках, заключаемых в qq и qx во встраиваемых документах (here documents). Наконец, qr обрабатывает постоянные участки в регулярных выражениях, как описано в конце главы 5 «Поиск по шаблону».

Обработчику передаются три аргумента. Первым аргументом является исходная константа в том виде, как она передается в Perl. Второй аргумент представляет то, как Perl фактически интерпретировал константу; например, 123_456 будет выглядеть как 123456.

Третий аргумент определяется только для строк, обрабатываемых обработчиками q и qr, и может иметь значения qq, q, s или tr в зависимости от того, как должна использоваться строка. qq означает, что строка взята из интерполируемого контекста, например, из двойных кавычек, обратных кавычек (backticks), соответствия m// или шаблона подстановки s///. q означает, что строка взята из не интерполированного контекста, значение s говорит о том, что строка является строкой замены в подстановке s///, а tr — что строка входит в состав выражения tr/// или y///.

Обработчик должен возвратить скаляр, который будет использоваться вместо константы. Часто этот скаляр является ссылкой на перегруженный объект, но ничто не мешает вам сделать что-либо более коварное:

```
package DigitDoubler;    # Модуль, помещаемый в DigitDoubler.pm
use overload;

sub import { overload::constant ( integer => \&handler,
                                  float   => \&handler ) }


sub handler {
    my ($orig, $interp, $context) = @_;
    return $interp * 2;          # удвоение всех констант
}

1;
```

Обратите внимание на то, что один обработчик используется двумя ключами, что в данном случае вполне оправданно. Теперь, сказав:

```
use DigitDoubler;

$trouble = 123;       # trouble теперь 246
$jeopardy = 3.21;     # jeopardy теперь 6.42
```

вы измените окружающий мир.

Если перехватываются строковые константы, рекомендуется также создать оператор конкатенации (".."), потому что интерполируемое выражение типа "ab$cd!!" является просто сокращенной записью более длинного 'ab' . $cd . '!!'. Аналогично, отрицательные числа рассматриваются как отрицания положительных констант, поэтому нужно создать обработчик для neg, если перехватываются целые или действительные числа. (Раньше нам это не требовалось, потому что мы возвращаем фактические числа, а не ссылки на перегруженные объекты.)

Обратите внимание, что overload::constant не распространяет свое действие на компиляцию внутри eval на этапе компиляции, что можно рассматривать

как ошибку или функциональную особенность (bug или feature) в зависимости от вашей точки зрения.

Открытые функции перегрузки

В версии Perl 5.6 прагма `use overload` предоставляет следующие функции для общественного потребления.

`overload::StrVal(OBJ)`

Эта функция возвращает строковое значение, которое должен иметь *OBJ* в отсутствие перегрузки строки (`""`).

`overload::Overloaded(OBJ)`

Эта функция возвращает значение true, если для *OBJ* действует какая-либо перегрузка операторов, и false в противном случае.

`overload::Method(OBJ, OPERATOR)`

Эта функция возвращает ссылку на код, реализующий перегрузку *OPERATOR*, когда тот выполняет действия над *OBJ*, или `undef`, если такой перегрузки не существует.

Наследование и перегрузка

Наследование взаимодействует с перегрузкой двумя путями. Во-первых, взаимодействие происходит, когда обработчик именуется строкой, а не указывается в виде ссылки на код или анонимной подпрограммы. При указании в виде строки обработчик интерпретируется как метод и может таким образом наследоваться у надклассов.

Второй путь взаимодействия наследования и перегрузки заключается в том, что любой класс, являющийся производным от перегруженного класса, в свою очередь подвергается этой перегрузке. Иными словами, перегрузка сама наследуется. Набор обработчиков для класса рекурсивно является объединением обработчиков для всех предков этого класса. Если обработчик есть в нескольких различных предках, то фактически используемый обработчик определяется обычными правилами наследования методов. Например, если класс `Alpha` наследует классам `Beta` и `Gamma` в указанном порядке и класс `Beta` перегружает + посредством `\&Beta::plus_sub`, а класс `Gamma` перегружает + строкой `"plus_meth"`, то при попытке применения + к объекту `Alpha` будет вызвана `Beta::plus_sub`.

Так как значением ключа `fallback` не является обработчик, его наследование не подчиняется указанным выше правилам. В текущей реализации используется значение `fallback` для первого перегруженного предка, но этот выбор случаен и может быть изменен без уведомления (по крайней мере, без особого уведомления).

Перегрузка на этапе исполнения

Поскольку команды `use` выполняются на этапе компиляции, единственный способ изменить перегрузку на этапе исполнения следующий:

```
eval " use overload '+' => \&my_add ";
```

Можно также сказать:

```
eval " no overload '+', '- -', '<=' ";
```

хотя использование таких конструкций на этапе исполнения спорно.

Диагностика перегрузки

Если Perl компилировался с ключом `-DDEBUGGING`, можно увидеть диагностические сообщения для перегрузки при запуске программы с ключом `-Do` или его эквивалентом. Можно также узнать, какие операции перегружены, с помощью команды `m` встроенного отладчика Perl.

Если вы сейчас чувствуете перегрузку, то следующая глава, возможно, расставит все по своим местам.

14

Связанные переменные

Некоторые человеческие начинания требуют маскировки. Иногда ее целью является обман, но чаще намерение заключается в передаче информации на более глубоком уровне. Например, приглашая соискателя на собеседование, интервьюер рассчитывает, что тот придет в галстуке, чтобы показать серьезную заинтересованность в получении работы, хотя оба знают, что он никогда не будет носить галстук на работе. Если вдуматься, то это странно: повязав на шею кусок тряпки можно чудесным образом получить работу. В культуре Perl оператор *tie* («связывать» или «галстук») играет схожую роль: он позволяет создавать обычную с виду переменную, которая, под своей личиной, оказывается полноценным объектом Perl со своей собственной интересной личностью. Это просто некоторая магия, как вытаскивание кролика из шляпы.

Скажем иначе, разыменовывающие символы `$`, `@`, `%` и `*` перед именем переменной многое сообщают Perl и программирующим на нем: каждый из них предполагает определенный набор первичных режимов поведения. Эти режимы можно преобразовать различными полезными способами с помощью `tie`, связав переменную с классом, реализующим новый набор режимов поведения. Например, можно создать обычный хеш Perl, а затем связать его (`tie`) с классом, превращающим хеш в базу данных, в результате чего при чтении данных из хеша Perl волшебным образом осуществляет выборку данных из внешнего файла базы данных, а при записи данных в хеш Perl волшебным образом сохраняет данные во внешнем файле базы данных. В данном случае «волшебным образом» означает «невидимым образом сделать что-то очень сложное». Как гласит старое изречение, любая достаточно развитая технология неотличима от сценария Perl. (Без шуток, для тех, кто

работает с внутренними механизмами Perl, *magic* (волшебство) это технический термин, означающий всякую дополнительную семантику применения таких переменных, как %ENV или %SIG. Связанные переменные являются для этого дополнительным расширением.)

В Perl уже есть встроенные функции dbmopen и dbmclose, которые волшебным образом связывают переменные типа хеш с базами данных, однако они остались с тех времен, когда в Perl не было tie. Сейчас tie предоставляет более общий механизм. В действительности сам Perl реализует dbmopen и dbmclose с помощью tie.

Можно связать скаляр, массив, хеш или указатель файла (через его typeglob) с любым классом, который предоставляет надлежащим образом названные методы, чтобы перехватывать и эмулировать обычный доступ к этим переменным. Первый из этих методов вызывается в месте вызова самого tie: связывание переменной всегда вызывает конструктор, который, в случае успешного выполнения, возвращает объект, который Perl складывает про запас туда, где он вам не виден, внутрь «обычной» переменной. Впоследствии всегда можно извлечь этот объект из обычной переменной с помощью функции tied:

```
tie VARIABLE, CLASSNAME, LIST;  # привязывает VARIABLE к CLASSNAME
$object = tied VARIABLE;
```

Эти две строки эквивалентны следующей:

```
$object = tie VARIABLE, CLASSNAME, LIST;
```

После привязки можно работать с обычной переменной как всегда, но при каждом обращении к ней автоматически вызываются методы лежащего в основе объекта; вся сложная организация класса скрывается за этими вызовами методов. Если впоследствии потребуется разорвать связь между переменной и классом, можно отвязать переменную:

```
untie VARIABLE;
```

Можно воспринимать tie почти как необычного вида bless, за исключением того, что «благословляется» голая переменная, а не ссылка на объект. Кроме того, функция tie может принимать дополнительные параметры, совсем как конструктор, что не очень удивляет, поскольку фактически она внутренне вызывает конструктор, имя которого зависит от типа связываемой переменной: TIESCALAR, TIEARRAY, TIEHASH или TIEHANDLE.[1] Эти конструкторы вызываются как методы класса с заданным *CLASSNAME* в качестве агента и дополнительными аргументами, указанными в *LIST*. (Переменная *VARIABLE* не передается конструктору.)

---

[1] Поскольку у конструкторов свои имена, можно даже создать единый класс, который реализует их все. Это позволит вам привязать скаляры, массивы, хеши и дескрипторы файлов к одному и тому же классу, хотя обычно так не делают, поскольку это затрудняет написание других магических методов.

Каждый из этих четырех конструкторов возвращает объект особым образом. Им, как и другим методам класса, безразлично, что они вызваны из tie, поскольку при желании их всегда можно вызвать непосредственно. В некотором смысле все волшебство заключено в tie, а не в классе, реализующем связывание. Что касается класса, то это обычный класс с необычными именами методов. (В действительности некоторые связанные модули предоставляют дополнительные методы, невидимые через связанную переменную; эти методы должны вызываться явно, как любые другие методы объекта. Такие дополнительные методы могут предоставлять дополнительные сервисы в виде блокировки файлов, защиты транзакций или чего-то иного, что может делать метод экземпляра.)

Поэтому данные конструкторы выполняют bless и возвращают ссылку на объект, как любой другой конструктор. Эта ссылка не обязательно должна ссылаться на переменную того же типа, что и связываемая; она только должна быть обработана bless, чтобы связанная переменная могла найти обратную дорогу к вашему классу в поисках поддержки. Например, в нашем длинном примере TIEARRAY будет использоваться объект на основе хеша, в котором удобно хранить дополнительную информацию об эмулируемом массиве.

Функция tie не загрузит вместо программиста модуль с помощью use или require: он должен, при необходимости, сделать это самостоятельно, прежде чем вызвать tie. (С другой стороны, функция dbmopen в целях обратной совместимости попытается загрузить через use ту или иную реализацию модуля базы данных. Однако можно заменить ее выбор с помощью явной команды use при условии, что загружаемый модуль есть в списке опробуемых функцией dbmopen. Более полное разъяснение можно найти в электронной документации по модулю AnyDBM_File.)

Методы, вызываемые связанной переменной, имеют предустановленные имена, например FETCH и STORE, поскольку они вызываются неявным образом (т. е. в ответ на некоторые события) изнутри Perl. Эти имена записываются целиком ЗАГЛАВНЫМИ БУКВАМИ, что является часто используемым соглашением для таких неявно вызываемых программ. (В число других специальных имен, следующих этому соглашению, входят BEGIN, CHECK, INIT, END, DESTROY и AUTOLOAD, а также UNIVERSAL->VERSION. На самом деле почти все предопределенные переменные и дескрипторы файлов в Perl пишутся на верхнем регистре: STDIN, SUPER, CORE, CORE::GLOBAL, DATA, @EXPORT, @INC, @ISA, @ARGV и %ENV. Встроенные операторы и прагмы ударяются в противоположную крайность и вообще не используют заглавные буквы.)

Начнем мы с того, что расскажем о самой простой вещи: о том, как связать скалярную переменную.

Связывание скаляров

Для осуществления связывания скаляров в классе должны быть определены следующие методы: TIESCALAR, FETCH и STORE (возможно, также DESTROY). При

связывании скалярной переменной Perl вызывает TIESCALAR. При чтении скалярной переменной он вызывает FETCH, а при присвоении значения этой переменной вызывает STORE. Если сохранить объект, возвращенный первоначальным обращением к tie (или позже извлечь его с помощью tied), то можно самостоятельно обращаться к лежащему в основе объекту: методы FETCH и STORE не будут при этом запускаться. В качестве объекта он не представляет собой ничего волшебного, скорее вполне объективное.

Если существует метод DESTROY, Perl вызывает его при исчезновении последней ссылки на связанный объект – как для всякого другого объекта. Это случается, когда программа завершает работу или вызывается untie, удаляющая ссылку, используемую tie. Однако untie не удаляет ожидающие обработки ссылки, которые могли быть помещены куда-то еще; DESTROY тоже откладывается до того времени, когда исчезнут эти ссылки.

Пакеты Tie::Scalar и Tie::StdScalar, которые находятся в стандартном модуле Tie::Scalar, предоставляют некоторые простые определения базового класса на тот случай, если вам не хочется определять эти методы самостоятельно. Tie::Scalar предоставляет простейшие методы, которые делают очень мало, а Tie::StdScalar предоставляет методы, благодаря которым связанный скаляр ведет себя как обычный скаляр Perl. (Что кажется совершенно бесполезным, но иногда требуется некоторая простая оболочка вокруг обычной семантики скаляров – например, чтобы подсчитать, сколько раз выполнялось присвоение некоторой переменной.)

Прежде чем демонстрировать наш детальный пример и описывать всю действующую в нем механику, дадим вам маленький кусочек – для возбуждения аппетита и чтобы показать, как просто все на самом деле. Вот программа целиком:

```
#!/usr/bin/perl
package Centsible;
sub TIESCALAR { bless \my $self, shift }
sub STORE { ${ $_[0] } = $_[1] }  # сделать дела по умолчанию
sub FETCH { sprintf "%.02f", ${ my $self = shift } } # округлить значение

package main;
tie $bucks, "Centsible";
$bucks = 45.00;
$bucks *= 1.0715; # налог
$bucks *= 1.0715; # удвоить налог!
print "С вас, пожалуйста, $bucks.\n";
```

При выполнении этой программы выводится:

```
С вас, пожалуйста, 51.67.
```

Чтобы увидеть разницу, закомментируйте вызов tie, и вы получите:

```
С вас, пожалуйста, 51.66505125.
```

Согласитесь, что работы здесь больше, чем при обычном округлении чисел.

Методы связывания скаляров

Теперь, когда вы на примере ознакомились с тем, что нам предстоит, разработаем более сложный класс для связывания скаляров. Вместо применения готового пакета для базового класса (в особенности потому, что скаляры столь просты), рассмотрим поочередно все четыре метода и создадим пример класса с именем ScalarFile. Скаляры, связанные с этим классом, содержат простые строки, а каждая такая переменная неявно связана с файлом, в котором хранится эта строка. (Можно назвать свои переменные так, чтобы они напоминали о том, на какой файл мы ссылаемся.) Переменные связываются с классом так:

```
use ScalarFile;        # Загрузить ScalarFile.pm
tie $camel, "ScalarFile", "/tmp/camel.lot";
```

При связывании переменной ее предшествующее содержание разрушается, а внутренняя связь между переменной и ее объектом замещает обычную семантику этой переменной. При запросе значения переменной теперь читается содержимое */tmp/camel.lot*, а в результате присвоения $camel значения в */tmp/camel.lot* записывается новое значение, уничтожающее предшествующее.

Происходит связывание с переменной, а не с ее значением, поэтому связанная природа переменной не распространяется путем присвоения. Допустим, например, что копируется связанная переменная:

```
$dromedary = $camel;
```

Вместо того чтобы обычным образом прочесть значение скалярной переменной $camel, Perl вызывает метод FETCH с лежащим в основе ассоциированным объектом. Все происходит так, как если бы было написано:

```
$dromedary = (tied $camel)->FETCH():
```

Если же запомнить объект, возвращенный tie, то можно непосредственно использовать эту ссылку, как в следующем примере:

```
$clot = tie $camel, "ScalarFile", "/tmp/camel.lot";
$dromedary = $camel;          # через неявный интерфейс
$dromedary = $clot->FETCH();  # то же, но явно
```

Если в классе есть другие методы помимо TIESCALAR, FETCH, STORE и DESTROY, то можно при помощи $clot вызвать их вручную. Однако обычно каждый занимается своим делом и не трогает лежащий в основе объект, поэтому можно часто видеть, что значение, возвращаемое tie, игнорируется. Тем не менее, можно получить объект через tied, если он понадобится позже (например, если в классе есть другие методы, которые вам нужны). Игнорирование возвращаемого объекта также устраняет некоторые виды ошибок, о которых рассказывается далее.

Вот преамбула нашего класса, которую мы поместим в *ScalarFile.pm*:

```
package ScalarFile;
use Carp;                # Распространение сообщений об ошибках.
use strict;              # Будем придерживаться некоторой дисциплины.
use warnings;            # Включение вывода предупреждений
                         # с лексической областью видимости.
use warnings::register;  # Разрешить пользователю сказать "use warnings
                         #'ScalarFile'".
my $count = 0;           # Внутренний счетчик связанных ScalarFile.
```

Стандартный модуль Carp экспортирует подпрограммы carp, croak и confess, которые будут использованы далее в примерах данного раздела. Как обычно, см. главу 32 «Стандартные модули» или электронную документацию по Carp.

В классе определены следующие методы.

CLASSNAME->TIESCALAR(*LIST*)

Метод класса TIESCALAR запускается всякий раз, когда вы связываете скалярную переменную. Необязательный *LIST* содержит параметры, необходимые для правильной инициализации объекта. (В нашем примере есть только один параметр: имя файла.) Метод должен возвращать объект, но это не обязательно должна быть ссылка на скаляр. Однако в нашем примере это так.

```
sub TIESCALAR {           # в ScalarFile.pm
    my $class    = shift;
    my $filename = shift;
    $count++;             # Лексическая переменная с областью видимости
                          # в файле, закрытая в классе.
    return bless \$filename, $class;
}
```

Поскольку нет скалярного эквивалента формирователей массива и хеша, [] и {}, мы просто передаем bless объект ссылки лексической переменной, который становится анонимным, как только имя выходит из области видимости. Это срабатывает (то же можно делать с массивами и хешами), если переменная действительно является лексической. Если попробовать проделать то же с глобальной переменной, то может показаться, что все проскочило, пока вы не попытаетесь создать еще один *camel.lot*. Не вздумайте написать что-нибудь подобное следующему:

```
sub TIESCALAR { bless \$_[1], $_[0] } # НЕВЕРНО, если ссылка на глобальную
                                        переменную.
```

Более надежно написанный конструктор может проверять доступность файла. Сначала мы проверим доступность файла по чтению, поскольку не хотим разрушить существующее значение. (Иными словами, не следует полагать, что пользователи сначала осуществляют запись. Они могут копить старые файлы Camel Lot от предыдущих прогонов программы.) Если нельзя открыть или создать файл с указанным именем, вежливо сообщим об ошибке, возвратив undef и, возможно, выведя предупреждение с по-

мощью carp. (Можно использовать для этого и croak, это дело вкуса.) Попробуем определить, интересно ли пользователю наше предупреждение, при помощи прагмы warnings:

```
sub TIESCALAR {           # в ScalarFile.pm
    my $class    = shift;
    my $filename = shift;
    my $fh;
    if (open $fh, "<", $filename or
        open $fh, ">", $filename)
    {
        close $fh;
        $count++;
        return bless \$filename, $class;
    }
    carp "Невозможно связать $filename: $!" if warnings::enabled();
    return;
}
```

Теперь, когда у нас есть такой конструктор, можно связать скаляр $string с файлом *camel.lot*:

```
tie ($string, "ScalarFile", "camel.lot") or die;
```

(Мы все еще принимаем некоторые допущения, чего делать не следовало. В промышленном варианте кода мы, вероятно, открыли бы указатель файла и запомнили его вместе с именем файла до конца действия связывания, заблокировав указатель на все время для монопольного использования с помощью flock. В противном случае мы не защищены от ситуации гонок – см. раздел «Ошибки синхронизации» главы 23 «Защита данных».)

SELF->FETCH

Этот метод вызывается при обращении к переменной (т. е. чтении ее значения). Он не принимает аргументов помимо объекта, связанного с переменной. В нашем примере объект содержит имя файла.

```
sub FETCH {
    my $self  = shift;
    confess "Я не метод класса" unless ref $self;
    return unless open my $fh, $$self;
    read($fh, my $value, -s $fh);  # NB: не используйте -s с каналами!
    return $value;
}
```

На этот раз мы решили выйти из себя (возбудить исключительную ситуацию), если FETCH возвратит что-либо, отличное от ссылки. (Когда он вызван как метод класса или по ошибке вызван как подпрограмма.) Другого способа возвратить ошибку нет, поэтому такой способ, вероятно, правильный. На самом деле Perl все равно возбудил бы исключительную ситуацию, как только мы попытались бы разыменовать $self; мы просто стараемся соблюдать вежливость и используем confess для того, чтобы

извергнуть всю последовательность вызовов из стека на экран пользователя. (Если это можно считать вежливостью.)

Теперь мы можем увидеть содержимое *camel.lot*, сказав следующее:

```
tie($string, "ScalarFile", "camel.lot");
print $string;
```

SELF->STORE(*VALUE*)

Этот метод выполняется, когда устанавливается переменная (ей присваивается значение). Первый аргумент, *SELF*, всегда является объектом, связанным с переменной; *VALUE* является тем, что присваивается переменной. (Мы вольно обращаемся с термином «присваивается» – любая операция, модифицирующая переменную, может вызвать STORE.)

```
sub STORE {
    my($self,$value) = @_;
    ref $self                   or confess "не метод класса";
    open my $fh, ">", $$self    or croak "нельзя разрушить $$self: $!";
    syswrite($fh, $value) == length $value
                                or croak "нельзя записать в $$self: $!";
    close $fh                   or croak "нельзя закрыть $$self: $!";
    return $value;
}
```

После «присвоения» мы возвращаем новое значение, поскольку это то, что делает присвоение. Если присвоение оказалось безуспешным, мы выводим сообщение посредством croak. Возможными причинами могут быть отсутствие разрешения на запись в ассоциированный файл, отсутствие места на диске или нападение гремлинов на контроллер диска. Иногда вы управляете волшебством, а иногда волшебство управляет вами.

Теперь мы так можем осуществить запись в *camel.lot*:

```
tie($string, "ScalarFile", "camel.lot");
$string  = "Вот первая строка camel.lot\n";
$string .= "А вот еще одна строка, дописанная автоматически.\n";
```

SELF->DESTROY

Этот метод запускается, когда объект, ассоциированный со связанной переменной, должен подвергнуться сборке мусора, если он должен сделать что-то особенное для уборки за собой. Как и для других классов, этот метод редко необходим, т. к. Perl автоматически освобождает память умирающего объекта. Сейчас мы определим метод DESTROY, который декрементирует счетчик связанных файлов:

```
sub DESTROY {
    my $self = shift;
    confess "неверный тип" unless ref $self;
    $count- -;
}
```

Можно также предоставить дополнительный метод класса для извлечения текущего значения счетчика. Фактически безразлично, вызывается он как метод класса или как метод объекта, но ведь после DESTROY у вас нет объекта, не так ли?

```
sub count {
    # my $invocant = shift;
    $count;
```

В любой момент можно вызвать его как метод класса:

```
if (ScalarFile->count) {
    warn "Где-то еще сидят связанные ScalarFiles...\n";
}
```

Это, пожалуй, все на данную тему. На самом деле это даже больше, поскольку мы сделали несколько хороших вещей, продиктованных соображениями полноты, надежности и общей эстетики (или отсутствия таковой). Конечно, можно создавать более простые классы TIESCALAR.

Переменные волшебных счетчиков

Вот простой класс Tie::Counter, вдохновленный одноименным модулем из CPAN. Переменные, связанные с этим классом, инкрементируются на 1 при каждом обращении к ним. Например:

```
tie my $counter, "Tie::Counter", 100;
@array = qw /Red Green Blue/;
for my $color (@array) {                # Выводит:
    print "  $counter  $color\n";        #   100  Red
}                                       #   101  Green
                                        #   102  Blue
```

Конструктор принимает в качестве необязательного аргумента начальное значение счетчика, по умолчанию равное 0. Присвоение счетчику устанавливает новое значение. Вот этот класс:

```
package Tie::Counter;
sub FETCH     { ++ ${ $_[0] } }
sub STORE     { ${ $_[0] } = $_[1] }
sub TIESCALAR {
    my ($class, $value) = @_;
    $value = 0 unless defined $value;
    bless \$value => $class;
}
1;  # если в модуле
```

Видите, какой он маленький? Чтобы построить такой класс, не требуется много кода.

Волшебное изгнание $_

Следующий любопытно экзотический класс служит для объявления вне закона нелокализованного использования $_. Вместо того чтобы ввести модуль с помощью команды use, которая вызывает метод класса import, этот модуль должен загружаться с помощью no для вызова редко применяемого метода unimport. Пользователь говорит:

```
no Underscore;
```

и после этого любое обращение к $_ как к нелокализованной глобальной переменной возбуждает исключительную ситуацию.

Вот маленький тестовый набор для модуля:

```
#!/usr/bin/perl
no Underscore;
@tests = (
    "Assignment"  => sub { $_ = "Bad" },
    "Reading"     => sub { print },
    "Matching"    => sub { $x = /badness/ },
    "Chop"        => sub { chop },
    "Filetest"    => sub { -x },
    "Nesting"     => sub { for (1..3) { print } },
);

while ( ($name, $code) = splice(@tests, 0, 2) ) {
    print "Testing $name: ";
    eval { &$code };
    print $@ ? "detected" : " missed!";
    print "\n";
}
```

что выводит следующее:

```
Testing Assignment: detected
Testing Reading: detected
Testing Matching: detected
Testing Chop: detected
Testing Filetest: detected
Testing Nesting: 123 missed!
```

В последнем случае «missed» получено благодаря должной локализации в цикле for и потому безопасному доступу.

А вот и сам любопытно-экзотический модуль Underscore. (Мы ведь говорили, что он любопытно-экзотический?) Он работает, поскольку связанное волшебство фактически скрыто с помощью local. Модуль выполняет tie в своем коде инициализации, поэтому require тоже работает.

```
package Underscore;
use Carp;
sub TIESCALAR { bless \my $dummy => shift }
```

```
sub FETCH { croak 'Доступ по чтению к $_ запрещен'  }
sub STORE { croak 'Доступ по записи к $_ запрещен' }
sub unimport { tie($_, __PACKAGE__) }
sub import   { untie $_ }
tie($_, __PACKAGE__) unless tied $_;
1;
```

Тяжело с пользой смешивать вызовы use и no для этого класса в вашей программе, поскольку все они происходят на этапе компиляции, а не на этапе исполнения. Можно было бы непосредственно вызвать Underscore->import и Underscore->unimport, как это делают use и no. Обычно, однако, чтобы сделать ренонс и позволить себе снова свободно использовать $_, следует применить к ней local, в чем и состоит все дело.

Связывание массивов

В классе, реализующем связанный массив, должны быть определены, по крайней мере, методы TIEARRAY, FETCH и STORE. Есть много необязательных методов: разумеется, вездесущий метод DESTROY, но также методы STORESIZE и FETCHSIZE, которые применяются для предоставления доступа к $#array и scalar(@array). Кроме того, когда Perl должен очистить массив, запускается CLEAR, а когда Perl должен предварительно расширить память в действительном массиве, то запускается EXTEND.

Можно также определить методы POP, PUSH, SHIFT, UNSHIFT, SPLICE, DELETE и EXISTS, если требуется, чтобы соответствующие функции Perl работали со связанным массивом. Класс Tie::Array может служить базовым для реализации первых пяти из этих функций на языке FETCH и STORE. (Реализация по умолчанию DELETE и EXISTS в Tie::Array просто вызывает croak.) Если FETCH и STORE определены, то не важно, какого типа структура данных содержится в вашем объекте.

С другой стороны, класс Tie::StdArray (определенный в стандартном модуле Tie::Array) предоставляет базовый класс с методами по умолчанию, которые предполагают, что объект содержит обычный массив. Вот простой класс для связывания массива, который это использует. Поскольку в качестве базового класса выступает Tie::StdArray, он должен определить только те методы, с которыми требуется нестандартное обращение.

```
#!/usr/bin/perl
package ClockArray;
use Tie::Array;
our @ISA = 'Tie::StdArray';
sub FETCH {
    my($self,$place) = @_;
    $self->[ $place % 12 ];
}
sub STORE {
    my($self,$place,$value) = @_;
```

```
    $self->[ $place % 12 ] = $value;
}

package main;
tie my @array, 'ClockArray';
@array = ( "a" ... "z" );
print "@array\n";
```

При прогоне программа выводит `"y z o p q r s t u v w x"`. Этот класс дает массив, у которого всего дюжина позиций, как у циферблата часов, пронумерованных от 0 до 11. Если запросить 15-й индекс массива, то будет получен в действительности 3-й. Считайте это помощью людям, которые не научились читать 24-часовой циферблат.

Методы связывания массивов

Это просто. Теперь некоторые обыденные подробности. Для их демонстрации мы создадим массив, границы которого фиксируются при его создании. Если попытаться обратиться к элементу за пределами этих границ, возбуждается исключительная ситуация. Например:

```
use BoundedArray;
tie @array, "BoundedArray", 2;

$array[0] = "отлично";
$array[1] = "хорошо";
$array[2] = "великолепно";
$array[3] = "тпру!";   # Запрещено; выводится сообщение об ошибке.
```

Код преамбулы для этого класса таков:

```
package BoundedArray;
use Carp;
use strict;
```

Чтобы в дальнейшем избежать необходимости определения `SPLICE`, мы унаследуем его от класса `Tie::Array`:

```
use Tie::Array;
our @ISA = ("Tie::Array");
```

CLASSNAME`->TIEARRAY(`*LIST*`)`

В качестве конструктора класса `TIEARRAY` должен возвращать освященную ссылку, через которую будет эмулироваться связанный массив.

В нашем следующем примере, чтобы показать, что в действительности не обязательно возвращать ссылку на массив, мы выбрали для представления нашего объекта ссылку на хеш. Хеш хорошо выполняет роль общего типа записей: значение в ключе хеша `BOUND` будет хранить максимально допустимую границу, а значение `DATA` будет содержать фактические данные. Если кто-то, находясь вне класса, попытается разыменовать возвра-

щаемый объект (несомненно, считая, что это ссылка на массив), возбуждается исключительная ситуация.

```
sub TIEARRAY {
    my $class = shift;
    my $bound = shift;
    confess "usage: tie(\@ary, 'BoundedArray', max_subscript)"
        if @_ || $bound =~ /\D/;
    return bless { BOUND => $bound, DATA => [] }, $class;
}
```

Теперь можно сказать:

```
tie(@array, "BoundedArray", 3);  # допускается максимальный индекс 3
```

и гарантировать, что в массиве будет не более четырех элементов. При чтении или записи отдельного элемента массива вызываются FETCH и STORE, как для скаляров, но с дополнительным аргументом, содержащим индекс.

SELF->FETCH(*INDEX*)

Этот метод запускается при обращении к отдельному элементу связанного массива. После объекта он получает один аргумент: индекс значения, выборку которого мы пытаемся осуществить.

```
sub FETCH {
    my ($self, $index) = @_;
    if ($index > $self->{BOUND}) {
        confess "Array OOB: $index > $self->{BOUND}";
    }
    return $self->{DATA}[$index];
}
```

SELF->STORE(*INDEX*, *VALUE*)

Этот метод вызывается, когда устанавливается элемент связанного массива. После объекта он получает два аргумента: индекс, по которому мы пытаемся что-то сохранить, и значение, которое мы пытаемся туда поместить. Например:

```
sub STORE {
    my($self, $index, $value) = @_;
    if ($index > $self->{BOUND} ) {
        confess "Array OOB: $index > $self->{BOUND}";
    }
    return $self->{DATA}[$index] = $value;
}
```

SELF->DESTROY

Perl вызывает этот метод, когда нужно уничтожить связанную переменную и освободить занимаемую ею память. Этого почти никогда не требуется в языке со сборкой мусора, поэтому в данном примере мы его опустим.

SELF->FETCHSIZE

Метод FETCHSIZE должен возвратить общее число элементов связанного массива, ассоциируемого с *SELF*. Его результат эквивалентен scalar(@array), что обычно равно $#array + 1.

```
sub FETCHSIZE {
    my $self = shift;
    return scalar @{$self->{DATA}};
}
```

SELF->STORESIZE(*COUNT*)

Этот метод устанавливает общее число элементов связанного массива, ассоциируемого с *SELF*, равным *COUNT*. Если массив сокращается, нужно удалить элементы за пределами *COUNT*. Если массив увеличивается, нужно обеспечить, чтобы новые позиции были неопределенными. Для нашего класса BoundedArray мы также гарантируем, что массив не выйдет за изначально установленные границы.

```
sub STORESIZE {
    my ($self, $count) = @_;
    if ($count > $self->{BOUND}) {
        confess "Array OOB: $count > $self->{BOUND}";
    }
    $#{$self->{DATA}} = $count;
}
```

SELF->EXTEND(*COUNT*)

С помощью метода EXTEND в Perl указывается, что массив, вероятно, будет расширен, чтобы вместить *COUNT* элементов. Благодаря этому можно выделить один большой участок памяти вместо того, чтобы впоследствии осуществлять много последовательных вызовов. Так как наши объекты BoundedArray имеют фиксированную верхнюю границу, мы не станем определять этот метод.

SELF->EXISTS(*INDEX*)

Этот метод проверяет существование в связанном массиве элемента в позиции *INDEX*. Для нашего класса BoundedArray мы просто используем встроенную функцию Perl exists после проверки, что это не попытка заглянуть дальше фиксированной верхней границы.

```
sub EXISTS  {
    my ($self, $index) = @_;
    if ($index > $self->{BOUND}) {
        confess "Array OOB: $index > $self->{BOUND}";
    }
    exists $self->{DATA}[$index];
}
```

SELF->DELETE(*INDEX*)

Метод DELETE удаляет элемент в позиции *INDEX* из связанного массива *SELF*. В нашем классе BoundedArray этот метод выглядит почти идентично EXISTS, но это отклонение от нормы.

```
sub DELETE {
    my ($self, $index) = @_;
    print STDERR "deleting!\n";
    if ($index > $self->{BOUND}) {
        confess "Array OOB: $index > $self->{BOUND}";
    }
    delete $self->{DATA}[$index];
}
```

SELF->CLEAR

Метод вызывается, когда нужно сделать массив пустым. Это происходит, когда массиву присваивается список новых значений (или пустой список), но не тогда, когда он передается функции undef. Поскольку очищенный BoundedArray всегда удовлетворяет условию верхней границы, ничего проверять здесь не нужно:

```
sub CLEAR {
    my $self = shift;
    $self->{DATA} = [];
}
```

Если список присваивается массиву, то CLEAR запустится, но значений списка не увидит. Поэтому при таком нарушении верхней границы:

```
tie(@array, "BoundedArray", 2);
@array = (1, 2, 3, 4);
```

возврат из метода CLEAR осуществляется успешно. Исключительная ситуация будет возбуждена при последующем выполнении STORE. Присвоение запускает один CLEAR и четыре STORE.

SELF->PUSH(*LIST*)

Этот метод добавляет в массив элементы из *LIST*. Вот как он может выглядеть для нашего класса BoundedArray:

```
sub PUSH    {
    my $self = shift;
    if (@_ + $#{$self->{DATA}} > $self->{BOUND}) {
        confess "Попытка протолкнуть слишком много элементов";
    }
    push @{$self->{DATA}}, @_;
}
```

SELF->UNSHIFT(*LIST*)

Этот метод помещает элементы *LIST* в начале массива. Для нашего класса BoundedArray эта подпрограмма аналогична PUSH.

SELF->POP

Метод POP удаляет из массива последний элемент и возвращает его. Для BoundedArray это метод-однострочник:

```
sub POP { my $self = shift; pop @{$self->{DATA}} }
```

SELF->SHIFT

Метод SHIFT удаляет из массива первый элемент и возвращает его. Для BoundedArray он аналогичен POP.

SELF->SPLICE(*OFFSET*, *LENGTH*, *LIST*)

Этот метод позволяет сращивать массив *SELF*. Для имитации splice — встроенной функции Perl — *OFFSET* является необязательным параметром, по умолчанию равным нулю, причем отрицательные значения ведут отсчет от конца массива. *LENGTH* тоже должна быть необязательной, по умолчанию задающей оставшуюся часть массива. *LIST* может быть пустым. Если метод правильно имитирует встроенную функцию, то должен возвратить список из исходных *LENGTH*-элементов, начинающийся с *OFFSET* (есть список элементов, заменяемых на LIST).

Поскольку сращивание представляет собой достаточно сложную операцию, мы вообще не будем ее определять; мы просто воспользуемся подпрограммой SPLICE из модуля Tie::Array, которая нам досталась в результате наследования Tie::Array. При этом SPLICE будет определена на основании других методов BoundedArray, поэтому проверка границ сохранится.

Этим завершается наш класс BoundedArray. Он лишь немного искажает семантику массивов. Но мы можем сделать и лучше, и гораздо более экономно.

Удобство обозначений

Одним из замечательных свойств переменных является возможность их интерполяции. Одним из не столь замечательных свойств функций является отсутствие возможности интерполяции. С помощью связанного массива можно создать интерполируемую функцию. Допустим, вы хотите вставлять в строку случайные числа. Можно просто сказать:

```
#!/usr/bin/perl
package RandInterp;
sub TIEARRAY { bless \my $self };
sub FETCH { int rand $_[1] };

package main;
tie @rand, "RandInterp";
    print "Случайным числом, меньшим $_, будет $rand[$_]\n";
}

$rand[32] = 5;    # Это переформатирует наш системный диск?
```

При прогоне этот код выводит:

```
Случайным числом, меньшим 1, будет 0
Случайным числом, меньшим 10, будет 3
Случайным числом, меньшим 100, будет 46
Случайным числом, меньшим 1000, будет 755
Can't locate object method "STORE" via package "RandInterp" at foo line 10.
[Невозможно найти метод объекта "STORE" через пакет "RandInterp"]
```

Как видите, нет большой беды, что мы даже не реализовали STORE. Просто, как обычно, возбуждается исключительная ситуация.

Связывание хешей

Класс, реализующий связанный хеш, должен определить восемь методов. TIEHASH создает новые объекты. FETCH и STORE осуществляют доступ к парам ключ/значение. EXISTS сообщает о том, присутствует ли ключ в хеше, а DELETE удаляет ключ вместе с его значением.[1] CLEAR очищает хеш, удаляя все пары ключ/значение. FIRSTKEY и NEXTKEY осуществляют обход пар ключ/значение, когда вызываются keys, values или each. И как всегда, если нужно осуществить какие-то особые действия при удалении объекта, можно определить метод DESTROY. (Если вам кажется, что методов очень много, значит, вы невнимательно прочли последний раздел, касающийся массивов. Во всяком случае можете смело наследовать методы по умолчанию из стандартного модуля Tie::Hash, переопределяя только те, которые вас интересуют. Кроме того, Tie::StdHash предполагает, что реализация тоже является хешем.)

Допустим, например, что нужно создать хеш, в котором при каждом присвоении значения ключу вместо переписывания прежнего значения новое значение дописывается к массиву значений. То есть, когда вы говорите:

```
$h{$k} = "один";
$h{$k} = "два";
```

на самом деле выполняется:

```
push @{ $h{$k} }, "один";
push @{ $h{$k} }, "два";
```

Это не очень сложная идея, поэтому для ее осуществления может использоваться довольно простой модуль. Если в качестве базового выступает класс Tie::StdHash, то так и будет. Вот класс Tie::AppendHash, который это реализует:

```
package Tie::AppendHash;
use Tie::Hash;
our @ISA = ("Tie::StdHash");
```

[1] Вспомните, что Perl делает различие между ключом, не существующим в хеше, и ключом существующим, но значением которого является undef. Проверить эти две возможности можно с помощью exists и defined соответственно.

```
sub STORE {
    my ($self, $key, $value) = @_;
    push @{$self->{key}}, $value;
}
1;
```

Методы связывания хешей

Вот пример интересного класса связанного хеша: он дает хеш, в котором представлены dot-файлы конкретного пользователя (т. е. файлы, имена которых начинаются с точки, как принято именовать файлы инициализации в Unix). Вы обращаетесь к хешу, передавая в качестве индекса имя файла (без точки), и получаете обратно содержимое соответствующего dot-файла. Например:

```
use DotFiles;
tie %dot, "DotFiles";
if ( $dot{profile} =~ /MANPATH/ or
     $dot{login}   =~ /MANPATH/ or
     $dot{cshrc}   =~ /MANPATH/    ) {
    print "Кажется, вам нужно установить переменную MANPATH\n";
}
```

Вот еще один способ использования нашего связанного класса:

```
# Третьим аргументом является имя пользователя, к dot-файлам которого мы привяжемся.
tie %him, "DotFiles", "daemon";
foreach $f (keys %him) {
    printf "dot-файл демона %s имеет размер %d\n", $f, length $him{$f};
}
```

В примере `DotFiles` мы реализуем объект как обычный хеш, содержащий несколько важных полей, из которых только поле `{CONTENTS}` содержит то, что пользователь считает хешем. Вот фактические поля объекта:

Поле	Содержание
`USER`	Чьи dot-файлы представляет объект.
`HOME`	Где находятся эти dot-файлы.
`CLOBBER`	Разрешено ли изменять и удалять эти dot-файлы.
`CONTENTS`	Хеш имен dot-файлов и отображений содержимого.

Вот начало *DotFiles.pm*:

```
package DotFiles;
use Carp;
sub whowasi { (caller(1))[3] . "()" }
my $DEBUG = 0;
sub debug { $DEBUG = @_ ? shift : 1 }
```

В данном примере мы хотим иметь возможность включить выдачу отладочной информации, которая облегчит трассировку во время разработки, для чего мы устанавливаем переменную $DEBUG. Мы также поддерживаем удобную внутреннюю функцию для облегчения вывода предупредительных сообщений: whowasi возвращает имя функции, вызвавшей текущую («дедушки» whowasi).

Вот методы связанного хеша DotFiles:

CLASSNAME->TIEHASH(*LIST*)

Вот конструктор DotFiles:

```
sub TIEHASH {
    my $self   = shift;
    my $user   = shift || $>;
    my $dotdir = shift || "";

    croak "usage: @{[ &whowasi ]} [USER [DOTDIR]]" if @_;

    $user = getpwuid($user) if $user =~ /^\d+$/;
    my $dir = (getpwnam($user))[7]
            or croak "@{ [&whowasi] }: no user $user";
    $dir .= "/$dotdir" if $dotdir;

    my $node = {
        USER        => $user,
        HOME        => $dir,
        CONTENTS    => {},
        CLOBBER     => 0,
    };

    opendir DIR, $dir
            or croak "@{[&whowasi]}: can't opendir $dir: $!";
    for my $dot ( grep /^\./ && -f "$dir/$_", readdir(DIR)) {
        $dot =~ s/^\.//;
        $node->{CONTENTS}{$dot} = undef;
    }
    closedir DIR;

    return bless $node, $self;
}
```

Стоит, вероятно, отметить, что если мы собираемся применять проверки файлов к значениям, возвращаемым readdir, то следует добавлять перед ними соответствующий каталог (как мы и делаем). Иначе, поскольку не выполнялась chdir, окажется, что мы проверяем не тот файл.

SELF->FETCH(*KEY*)

Этот метод реализует чтение элемента из связанного хеша. Вслед за объектом он принимает один аргумент: ключ, значение которого мы хотим получить. Ключ является строкой, и с ней можно делать что угодно (не противоречащее тому, что это строка).

Вот пример выборки для нашего DotFiles:

```
sub FETCH {
    carp &whowasi if $DEBUG;
    my $self = shift;
    my $dot = shift;
    my $dir = $self->{HOME};
    my $file = "$dir/.$dot";

    unless (exists $self->{CONTENTS}->{$dot} || -f $file) {
        carp "@{[&whowasi]}: no $dot file" if $DEBUG;
        return undef;
    }

    # Реализовать хеш.
    if (defined $self->{CONTENTS}->{$dot}) {
        return $self->{CONTENTS}->{$dot};
    } else {
        return $self->{CONTENTS}->{$dot} = `cat $dir/.$dot`;
    }
}
```

Мы несколько смошенничали, выполнив команду Unix *cat*(1), потому что более переносимым (и более эффективным) было бы самостоятельное открытие файла. С другой стороны, поскольку dot-файлы являются понятием Unix, мы не очень этим озабочены.

SELF->STORE(*KEY*, *VALUE*)

Этот метод делает всю черную работу, когда устанавливается элемент в связанном хеше (производится запись). Вслед за объектом он принимает два аргумента: ключ, с которым мы записываем новое значение, и само это значение.

В нашем примере DotFiles мы разрешим пользователям перезаписать файл лишь после вызова метода clobber с исходным объектом, возвращенным tie:

```
sub STORE {
    carp &whowasi if $DEBUG;
    my $self  = shift;
    my $dot   = shift;
    my $value = shift;
    my $file  = $self->{HOME} . "/.$dot";

    croak "@{[&whowasi]}: $file not clobberable"
        unless $self->{CLOBBER};

    open(F, "> $file") or croak "не могу открыть $file: $!";
    print F $value;
    close(F);
}
```

Если кому-то нужно что-то разрушить, можно сказать:

```
$ob = tie %daemon_dots, "daemon";
$ob->clobber(1);
$daemon_dots{signature} = "Настоящий демон \n";
```

Другим вариантом является установка {CLOBBER} посредством tied:

```
tie %daemon_dots, "DotFiles", "daemon";
tied(%daemon_dots)->clobber(1);
```

или в виде одной команды:

```
(tie %daemon_dots, "DotFiles", "daemon")->clobber(1);
```

Метод clobber определяется просто как:

```
sub clobber {
    my $self = shift;
    $self->{CLOBBER} = @_ ? shift : 1;
}
```

SELF->DELETE(*KEY*)

Этот метод обрабатывает запросы на удаление из хеша элемента. Если ваш эмулированный хеш использует действительный хеш, можете вызвать действительный delete. Кроме того, мы постараемся проверить, действительно ли пользователь хочет разрушить файлы:

```
sub DELETE   {
    carp &whowasi if $DEBUG;
    my $self = shift;
    my $dot = shift;
    my $file = $self->{HOME} . "/.$dot";
    croak "@{[&whowasi]}: не стану удалять файл $file"
        unless $self->{CLOBBER};
    delete $self->{CONTENTS}->{$dot};
    unlink $file or carp "@{[&whowasi]}: не могу удалить $file: $!";
}
```

SELF->CLEAR

Этот метод выполняется, когда нужно полностью очистить хеш, обычно в случае присвоения ему пустого списка. В нашем примере при этом удаляются все dot-файлы пользователя! Это настолько опасно, что мы потребуем, чтобы значение CLOBBER было больше 1, прежде чем выполнить это действие:

```
sub CLEAR {
    carp &whowasi if $DEBUG;
    my $self = shift;
    croak "@{[&whowasi]}: не буду удалять все dot-файлы $self->{USER}"
        unless $self->{CLOBBER} > 1;
```

```
        for my $dot ( keys %{$self->{CONTENTS}}) {
            $self->DELETE($dot);
        }
    }
```

SELF->EXISTS(*KEY*)

Этот метод запускается при вызове пользователем функции exists с некоторым хешем. В нашем примере мы будем искать ответ в элементе хеша {CONTENTS}:

```
sub EXISTS   {
    carp &whowasi if $DEBUG;
    my $self = shift;
    my $dot  = shift;
    return exists $self->{CONTENTS}->{$dot};
}
```

SELF->FIRSTKEY

Этот метод вызывается, когда пользователь начинает обход хеша, например, при вызове keys, values или each. В результате вызова keys в скалярном контексте мы сбрасываем ее внутреннее состояние, чтобы очередной each в команде return возвращал первый ключ.

```
sub FIRSTKEY {
    carp &whowasi if $DEBUG;
    my $self = shift;
    my $temp = keys %{$self->{CONTENTS}};
    return scalar each %{$self->{CONTENTS}};
}
```

SELF->NEXTKEY(*PREVKEY*)

Этот метод служит итератором для функций keys, values или each. *PREVKEY* является последним ключом, к которому производилось обращение, и который Perl умеет передавать. Это удобно, если методу NEXTKEY нужно знать свое предыдущее состояние для вычисления последующего.

В нашем примере для представления данных связанного хеша используется реальный хеш, только этот хеш хранится в поле CONTENTS хеша, а не в самом хеше. Поэтому мы можем просто положиться на итератор Perl each:

```
sub NEXTKEY  {
    carp &whowasi if $DEBUG;
    my $self = shift;
    return scalar each %{ $self->{CONTENTS} }
}
```

SELF->DESTROY

Этот метод запускается, когда объект связанного хеша должен быть удален из памяти. Практической нужды в нем нет, за исключением задач отладки и дополнительной подчистки. Вот очень простой вариант:

```
sub DESTROY  {
    carp &whowasi if $DEBUG;
}
```

Теперь, когда мы показали все эти методы, вот вам домашнее задание: вернуться назад, найти все места, в которых мы интерполировали `@{[&whowasi]}`, и заменить их простым связанным скаляром с именем `$whowasi`, осуществляющим то же самое.

Связывание указателей файлов

Класс, реализующий связанный указатель файла, должен определить следующие методы: `TIEHANDLE` и, по крайней мере, один из `PRINT`, `PRINTF`, `WRITE`, `READLINE`, `GETC` и `READ`. Класс может также предоставить метод `DESTROY` и методы `BINMODE`, `OPEN`, `CLOSE`, `EOF`, `FILENO`, `SEEK`, `TELL`, `READ` и `WRITE`, чтобы разрешить применение соответствующих встроенных функций Perl со связанным указателем файла. (На самом деле это не совсем точно: `WRITE` соответствует `syswrite` и не имеет никакого отношения к встроенной функции Perl `write` для вывода с объявлениями форматов.)

Связанные указатели файлов особенно удобны, когда Perl встраивается в другую программу (например, Apache или *vi*) и вывод на `STDOUT` или `STDERR` нужно переадресовать каким-либо особым образом.

Однако указатели файлов совсем не обязательно связывать с файлами. Команды вывода можно использовать для создания структур данных в памяти, а команды ввода – для чтения их в обратном направлении. Вот простой способ обратить последовательность команд `print` и `printf`, не обращая при этом отдельных строчек:

```
package ReversePrint;
use strict;
sub TIEHANDLE {
    my $class = shift;
    bless [], $class;
}
sub PRINT {
    my $self = shift;
    push @$self, join '', @_;
}
sub PRINTF {
    my $self = shift;
    my $fmt = shift;
    push @$self, sprintf $fmt, @_;
}
sub READLINE {
    my $self = shift;
    pop @$self;
}
```

```
package main;
my $m = "- -MORE- -\n";
tie *REV, "ReversePrint";

# Выполнить несколько print и printf.
print REV "The fox is now dead.$m";
printf REV <<"END", int rand 10000000;
The quick brown fox jumps over
over the lazy dog %d times!
END

print REV <<"END";
The quick brown fox jumps
over the lazy dog.
END

# Теперь обратное чтение из того же дескриптора.
print while <REV>;
```

В результате выводится:

```
The quick brown fox jumps
over the lazy dog.
The quick brown fox jumps over
over the lazy dog 3179357 times!
The fox is now dead.- -MORE- -
```

Методы привязки файлов

Для расширенного примера мы создадим указатель файла, который переводит на верхний регистр выводимые в этот файл строки. Исключительно для удовольствия мы будем начинать файл с <SHOUT> при его открытии и заканчивать его с </SHOUT> при закрытии. Благодаря этому мы можем выражаться на корректном (well-formed) XML.

Вот начало файла *Shout.pm*, который будет реализовывать класс:

```
package Shout;
use Carp;                # Чтобы выводить ошибки посредством croak
```

Теперь перечислим определения методов в *Shout.pm*.

CLASSNAME->TIEHANDLE(*LIST*)

Это конструктор класса, возвращающий, как обычно, обработанную bless ссылку.

```
sub TIEHANDLE {
    my $class = shift;
    my $form = shift;
    open my $self, $form, @_   or croak "не могу открыть $form@_: $!";
    if ($form =~ />/) {
        print $self  "<SHOUT>\n";
```

```
        $$self->{WRITING} = 1;      # Не забудьте закрывающий тег
    }
    return bless $self, $class;     # $self является ссылкой на glob
}
```

Здесь мы открываем новый указатель файла в соответствии с режимом и именем файла, переданными оператору `tie`, выводим в файл `<SHOUT>` и возвращаем обработанную bless ссылку на него. В этой команде `open` много чего происходит, но мы отметим лишь, что в дополнение к «open or die», обычной идиоме, `my $self` предоставляет `open` неопределенный скаляр, который тот умеет самооживлять в typeglob. Тот факт, что это typeglob, тоже важен, потому что typeglob содержит не только действительный объект ввода/вывода файла, но и другие удобные структуры данных, получаемые попутно, например скаляр (`$$$self`), массив (`@$$self`) и хеш (`%$$self`). (Не будем говорить о подпрограмме, `&$$self`.)

`$form` является аргументом, содержащим имя файла или режим. Если это имя файла, то массив `@_` пуст и выполняется open с двумя аргументами. В противном случае `$form` указывает режим для остальных аргументов.

После open мы проверяем, нужно ли выводить открывающий тег. Если да, то мы это делаем. И сразу после этого мы используем одну из упомянутых нами глобальных структур данных. `$$self->{WRITING}` дает пример использования глобальных переменных для хранения интересной информации. В данном случае мы запоминаем, выводился ли открывающий тег, чтобы знать, нужно ли создавать соответствующий закрывающий тег. Мы используем хеш `%$$self`, чтобы дать полю приличествующее имя. Можно было бы использовать скаляр `$$$self`, но это не было бы самодокументацией. (Или было бы *только* само (self) документацией, в зависимости от того, как на это взглянуть.)

`SELF->PRINT(LIST )`

Этот метод реализует вывод связанного указателя. *`LIST`* содержит то, что передается `print`. В нашем методе все элементы *`LIST`* переводятся в верхний регистр:

```
sub PRINT {
    my $self = shift;
    print $self map {uc} @_;
}
```

`SELF``->READLINE`

Этот метод поставляет данные, когда чтение из указателя осуществляется через оператор угловых скобок (`<FH>`) или `readline`. Метод должен возвращать `undef`, если данных больше нет.

```
sub READLINE {
    my $self = shift;
    return <$self>;
}
```

Здесь мы просто возвращаем <$self>, чтобы метод вел себя должным образом при вызове в скалярном или списковом контексте.

SELF->GETC

Этот метод выполняется, когда со связанным дескриптором файла используется getc.

```
sub GETC {
    my $self = shift;
    return getc($self);
}
```

Как и некоторые другие методы нашего класса Shout, метод GETC просто вызывает соответствующую встроенную функцию Perl и возвращает результат.

SELF->OPEN(*LIST*)

Наш метод TIEHANDLE сам открывает файл, но программа, использующая класс Shout, которая вызывает потом open, запускает этот метод.

```
sub OPEN {
    my $self = shift;
    my $form = shift;
    my $name = "$form@_";
    $self->CLOSE;
    open($self, $form, @_)      or croak "невозможно вновь открыть $name: $!";
    if ($form =~ />/) {
        print $self "<SHOUT>\n" or croak "невозможно начать вывод: $!";
        $$self->{WRITING} = 1;     # Запомнить, что нужно вывести закрывающий тег
    }
    else {
       $$self->{WRITING} = 0;     # Запомнить, что не нужно выводить закрывающий
тег
    }
    return 1;
}
```

Мы вызываем собственный метод CLOSE, чтобы явно закрыть файл, если пользователь об этом не позаботился. Затем мы открываем новый файл с именем, указанным в open, и напускаем на него shout.

SELF->CLOSE

Этот метод обрабатывает запрос на закрытие файла. Мы ищем конец файла и в случае успеха выводим </SHOUT>, а затем вызываем встроенную функцию Perl close.

```
sub CLOSE {
    my $self = shift;
    if ($$self->{WRITING}) {
        $self->SEEK(0, 2)             or return;
        $self->PRINT("</SHOUT>\n")    or return;
```

```
    }
    return close $self;
}
```

SELF->SEEK(*LIST*)

При выполнении seek в связанном указателе файла вызывается метод SEEK.

```
sub SEEK {
    my $self = shift;
    my ($offset, $whence) = @_;
    return seek($self, $offset, $whence);
}
```

SELF->TELL

Этот метод вызывается при использовании tell со связанным указателем.

```
sub TELL {
    my $self = shift;
    return tell $self;
}
```

SELF->PRINTF(*LIST*)

Этот метод запускается, когда со связанным указателем файла используется printf. Список *LIST* содержит формат и подлежащие выводу элементы.

```
sub PRINTF {
    my $self = shift;
    my $template = shift;
    return $self->PRINT(sprintf $template, @_);
}
```

Здесь мы применяем sprintf для генерации форматированной строки и передачи ее PRINT для перевода в верхний регистр. Однако ничто не заставляет вас использовать встроенную функцию sprintf. Управляющие символы процента можно интерпретировать по своему усмотрению.

SELF->READ(*LIST*)

Этот метод откликается, когда производится чтение из указателя с использованием read или sysread. Обратите внимание, что мы модифицируем первый аргумент, входящий в *LIST*, «по месту», подражая способности read заполнять скаляр, передаваемый как второй аргумент.

```
sub READ {
    my ($self, undef, $length, $offset) = @_;
    my $bufref = \$_[1];
    return read($self, $$bufref, $length, $offset);
}
```

SELF->WRITE(*LIST*)

Этот метод вызывается, когда производится запись в указатель с помощью syswrite. Здесь мы переводим строку, которую нужно записать, в верхний регистр.

```
sub WRITE {
    my $self = shift;
    my $string = uc(shift);
    my $length = shift || length $string;
    my $offset = shift || 0;
    return syswrite $self, $string, $length, $offset;
}
```

SELF->EOF

Этот метод возвращает булево значение, когда указатель файла, связанный с классом Shout, проверяется на состояние конца файла с использованием eof.

```
sub EOF {
    my $self = shift;
    return eof $self;
}
```

SELF->BINMODE(*DISC*)

Этот метод задает дисциплину ввода/вывода, которая должна использоваться с указателем файла. Если она не задана, то указатель файла переводится в двоичный режим (дисциплина :raw), если файловая система проводит различие между текстовыми и двоичными файлами.

```
sub BINMODE {
    my $self = shift;
    my $disc = shift || ":raw";
    return binmode $self, $disc;
}
```

Так следовало написать, но на практике в нашем случае это бесполезно, поскольку open уже произвел запись в указатель. Поэтому в данном случае следовало бы, вероятно, сказать:

```
sub BINMODE { croak("Слишком поздно использовать binmode") }
```

SELF->FILENO

Этот метод должен возвратить дескриптор файла (fileno), ассоциированный со связанным указателем файла операционной системой.

```
sub FILENO {
    my $self = shift;
    return fileno $self;
}
```

SELF->DESTROY

Как и при других типах связывания, этот метод запускается, когда связанный объект должен быть уничтожен. Он полезен для того, чтобы позволить объекту убрать за собой. В нем мы обеспечиваем закрытие файла, если программа забыла его закрыть. Мы могли бы просто сказать close $self, но лучше вызвать метод класса CLOSE. Тогда, если разработчик класса решит изменить способ закрытия файлов, этот метод DESTROY не нужно будет модифицировать.

```
sub DESTROY {
    my $self = shift;
    $self->CLOSE;      # Закрыть файл с помощью метода CLOSE класса Shout.
}
```

Вот демонстрация класса Shout:

```
#!/usr/bin/perl
use Shout;
tie(*FOO, Shout::, ">filename");
print FOO "hello\n";            # Выводит HELLO.
seek FOO, 0, 0;                 # Откат к началу.
@lines = <FOO>;                 # Вызывает метод READLINE.
close FOO;                      # Закрыть файл явно.
open(FOO, "+<", "filename");    # Переоткрыть FOO, вызвав OPEN.
seek(FOO, 8, 0);                # Пропустить "<SHOUT>\n".
sysread(FOO, $inbuf, 5);        # Прочесть 5 байтов из FOO в $inbuf.
print "found $inbuf\n";         # Должен вывести"hello".
seek(FOO, -5, 1);               # Вернуться назад к "hello".
syswrite(FOO, "ciao!\n", 6);    # Записать 6 байтов в FOO.
untie(*FOO);                    # Неявно вызывает метод CLOSE.
```

После выполнения этого кода файл содержит:

```
<SHOUT>
CIAO!
</SHOUT>
```

Вот еще некоторые странные и удивительные вещи, относящиеся к этой внутренней глобальной переменной. Мы используем тот же хеш, что и раньше, но с новыми ключами PATHNAME и DEBUG. Сначала мы установим перегрузку преобразования в строку, чтобы при выводе наших объектов отображался путь (см. главу 13 «Перегрузка»):

```
# Ну, это круто!
use overload q("") => sub { $_[0]->pathname };

# Это заглушка, помещаемая во все трассируемые функции.
sub trace {
    my $self = shift;
```

```
    local $Carp::CarpLevel = 1;
    Carp::cluck("\ntrace magical method") if $self->debug;
}

# Перегрузка обработчика для вывода строки.
sub pathname {
    my $self = shift;
    confess "я не метод класса" unless ref $self;
    $$self->{PATHNAME} = shift if @_;
    return $$self->{PATHNAME};
}

# Два режима.
sub debug {
    my $self = shift;
    my $var = ref $self ? \$$self->{DEBUG} : \our $Debug;
    $$var = shift if @_;
    return ref $self ? $$self->{DEBUG} || $Debug : $Debug;
}
```

А затем будем вызывать `trace` **при входе во все обычные методы, например:**

```
sub GETC { $_[0]->trace;           # НОВИНКА
    my($self) = @_;
    getc($self);
}
```

А также установим имя пути в `TIEHANDLE` **и** `OPEN`**:**

```
sub TIEHANDLE {
    my $class    = shift;
    my $form = shift;
    my $name = "$form@_";          # НОВИНКА
    open my $self, $form, @_   or croak "невозможно открыть $name: $!";
    if ($form =~ />/) {
        print $self  "<SHOUT>\n";
        $$self->{WRITING} = 1;     # Запомнить, что нужно вывести конечный тег
    }
    bless $self, $class;           # $fh является ссылкой на glob
    $self->pathname($name);        # НОВИНКА
    return $self;
}

sub OPEN { $_[0]->trace;           # НОВИНКА
    my $self = shift;
    my $form = shift;
    my $name = "$form@_";
    $self->CLOSE;
    open($self, $form, @_)      or croak "невозможно вновь открыть: $!";
    $self->pathname($name);        # НОВИНКА
    if ($form =~ />/) {
        print $self "<SHOUT>\n" or croak "невозможно начать вывод: $!";
        $$self->{WRITING} = 1;     # Запомнить, что нужно вывести конечный тег
```

```
    }
    else {
        $$self->{WRITING} = 0;     # Запомнить, что не нужно выводить конечный тег
    }
    return 1;
}
```

В каком-то месте нужно также вызвать `$self->debug(1)` для включения отладки. После этого все вызовы `Carp::cluck` будут выдавать осмысленные сообщения. Ниже приведено то, что мы получим во время повторного открытия файла, показанного выше. Видны три вызова методов вглубь при закрытии старого файла и подготовке к открытию нового:

```
trace magical method at foo line 87
    Shout::SEEK('>filename', '>filename', 0, 2) called at foo line 81
    Shout::CLOSE('>filename') called at foo line 65
    Shout::OPEN('>filename', '+<', 'filename') called at foo line 141
```

Созидающие указатели файлов

Один и тот же указатель файла может быть связан как с вводом, так и с выводом двунаправленного канала. Допустим, что вы хотите запустить программу *bc*(1) (калькулятор произвольной точности) таким способом:

```
use Tie::Open2;

tie *CALC, 'Tie::Open2', "bc -l";
$sum = 2;
for (1 .. 7) {
    print CALC "$sum * $sum\n";
    $sum = <CALC>;
    print "$_: $sum";
    chomp $sum;
}
close CALC;
```

Можно предположить, что будет выведено следующее:

```
1: 4
2: 16
3: 256
4: 65536
5: 4294967296
6: 18446744073709551616
7: 340282366920938463463374607431768211456
```

Ожидания были бы оправданы, если бы на компьютере имелась программа *bc*(1) и `Tie::Open2`, определенный так, как описано ниже. На этот раз в качестве внутреннего объекта будет использоваться массив. Он содержит два наши реальные указателя файлов для чтения и записи. (Черную работу от-

крытия двунаправленного канала выполняет метод `IPC::Open2`; мы занимаемся только приятными вещами.)

```
package Tie::Open2;
use strict;
use Carp;
use Tie::Handle;  # не наследуйте этому классу!
use IPC::Open2;

sub TIEHANDLE {
    my ($class, @cmd) = @_;
    no warnings 'once';
    my @fhpair = \do { local(*RDR, *WTR) };
    bless $_, 'Tie::StdHandle' for @fhpair;
    bless(\@fhpair => $class)->OPEN(@cmd) || die;
    return \@fhpair;
}

sub OPEN {
    my ($self, @cmd) = @_;
    $self->CLOSE if grep {defined} @{ $self->FILENO };
    open2(@$self, @cmd);
}

sub FILENO {
    my $self = shift;
    [ map { fileno $self->[$_] } 0,1 ];
}

for my $outmeth ( qw(PRINT PRINTF WRITE) ) {
    no strict 'refs';
    *$outmeth = sub {
        my $self = shift;
        $self->[1]->$outmeth(@_);
    };
}
for my $inmeth ( qw(READ READLINE GETC) ) {
    no strict 'refs';
    *$inmeth = sub {
        my $self = shift;
        $self->[0]->$inmeth(@_);
    };
}
for my $doppelmeth ( qw(BINMODE CLOSE EOF)) {
    no strict 'refs';
    *$doppelmeth = sub {
        my $self = shift;
        $self->[0]->$doppelmeth(@_) && $self->[1]->$doppelmeth(@_);
    };
}
for my $deadmeth ( qw(SEEK TELL)) {
    no strict 'refs';
    *$deadmeth = sub {
```

```
        croak("can't $deadmeth a pipe");
    };
}
1;
```

Последние четыре цикла, по нашему мнению, просто чрезвычайно элегантны. За разъяснением происходящего вернитесь к разделу, озаглавленному «Замыкания в качестве шаблонов функций» главы 8 «Ссылки».

Вот еще более безумный набор классов. Имена пакетов должны дать ключ к определению их назначения.

```
use strict;
package Tie::DevNull;

    sub TIEHANDLE {
        my $class = shift;
        my $fh = local *FH;
        bless \$fh, $class;
    }
    for (qw(READ READLINE GETC PRINT PRINTF WRITE)) {
        no strict 'refs';
        *$_  = sub { return };
    }

package Tie::DevRandom;

    sub READLINE { rand() . "\n"; }
    sub TIEHANDLE {
        my $class = shift;
        my $fh = local *FH;
        bless \$fh, $class;
    }
    sub FETCH { rand() }
    sub TIESCALAR {
        my $class = shift;
        bless \my $self, $class;
    }

package Tie::Tee;

    sub TIEHANDLE {
        my $class = shift;
        my @handles;
        for my $path (@_) {
            open(my $fh, ">$path") || die "невозможно вывести $path";
            push @handles, $fh;
        }
        bless \@handles, $class;
    }

    sub PRINT {
        my $self = shift;
        my $ok = 0;
        for my $fh (@$self) {
```

```
            $ok += print $fh @_;
        }
        return $ok == @$self;
    }
```

Класс `Tie::Tee` эмулирует стандартную программу Unix *tee*(1), которая отправляет один выходной поток по нескольким различным адресам. Класс `Tie::DevNull` эмулирует фиктивное устройство */dev/null* в Unix-системах. А класс `Tie::DevRandom` создает случайные числа в виде указателя или скаляра, в зависимости от того, вызываете вы `TIEHANDLE` или `TIESCALAR`! Вот как осуществляется обращение к ним:

```
package main;

tie *SCATTER,   "Tie::Tee", qw(tmp1 - tmp2 >tmp3 tmp4);
tie *RANDOM,    "Tie::DevRandom";
tie *NULL,      "Tie::DevNull";
tie my $randy,  "Tie::DevRandom";

for my $i (1..10) {
    my $line = <RANDOM>;
    chomp $line;
    for my $fh (*NULL, *SCATTER) {
        print $fh "$i: $line $randy\n";
    }
}
```

При этом на экран выводится что-нибудь вроде следующего:

```
1: 0.124115571686165 0.20872819474074
2: 0.156618299751194 0.678171662366353
3: 0.799749050426126 0.300184963960792
4: 0.599474551447884 0.213935286029916
5: 0.700232143543861 0.800773751296671
6: 0.201203608274334 0.0654303290639575
7: 0.605381294683365 0.718162304090487
8: 0.452976481105495 0.574026269121667
9: 0.736819876983848 0.391737610662044
10: 0.518606540417331 0.381805078272308
```

Но это не все! Вывод на экран был произведен благодаря «-» в `*SCATTER tie`. Но в той же строке было потребовано создать файлы *tmp1*, *tmp2* и *tmp4*, а также произвести дописывание к файлу *tmp3*. (Мы также осуществляли в цикле вывод в дескриптор файла `*NULL`, хотя он, конечно, ни в каких представляющих интерес местах не проявился – если, конечно, для вас не представляют интерес черные дыры.)

Тонкая ловушка при отвязывании

Когда используется объект, возвращаемый `tie` или `tied`, а в классе определен деструктор, то есть тонкая ловушка, от которой вы должны уберегаться.

Рассмотрим такой (возможно, натянутый) пример класса, использующего файл для регистрации всех значений, присваиваемый скаляру:

```
package Remember;

sub TIESCALAR {
    my $class = shift;
    my $filename = shift;
    open(my $handle, ">", $filename)
         or die "Невозможно открыть $filename: $!\n";
    print $handle "Старт\n";
    bless {FH => $handle, VALUE => 0}, $class;
}

sub FETCH {
    my $self = shift;
    return $self->{VALUE};
}

sub STORE {
    my $self = shift;
    my $value = shift;
    my $handle = $self->{FH};
    print $handle "$value\n";
    $self->{VALUE} = $value;
}

sub DESTROY {
    my $self = shift;
    my $handle = $self->{FH};
    print $handle "Конец\n";
    close $handle;
}
1;
```

Вот пример, в котором используется наш класс `Remember`:

```
use strict;
use Remember;

my $fred;
$x = tie $fred, "Remember", "camel.log";
$fred = 1;
$fred = 4;
$fred = 5;
untie $fred;
system "cat camel.log";
```

А вот вывод, получаемый при его выполнении:

```
Старт
1
4
5
Конец
```

Пока все хорошо. Добавим в класс Remember еще один метод, разрешающий комментарии в файле, например, такой:

```
sub comment {
    my $self = shift;
    my $message = shift;
    print { $self->{FH} } $handle $message, "\n";
}
```

И вот предыдущий пример, измененный для использования метода comment:

```
use strict;
use Remember;

my ($fred, $x);
$x = tie $fred, "Remember", "camel.log";
$fred = 1;
$fred = 4;
comment $x "changing...";
$fred = 5;
untie $fred;
system "cat camel.log";
```

Теперь файл окажется пустым, чего вы, вероятно, не желали. И вот почему. Связывание переменной ассоциирует ее с объектом, возвращаемым конструктором. На этот объект обычно имеется только одна ссылка: та, которая скрывается за самой связанной переменной. Вызов untie разрывает ассоциацию и ликвидирует эту ссылку. Поскольку ссылок на объект больше не осталось, запускается метод DESTROY.

Однако в вышеприведенном примере мы имели вторую ссылку на объект, связанную с $x. Это значит, что после untie по-прежнему сохраняется действующая ссылка на объект. DESTROY запущен не будет, записи буфера в файл и закрытия его не произойдет. Вот почему не было выдачи: буфер указателя файла по-прежнему находился в памяти. На диск он не попадет до выхода из программы.

Для обнаружения этого можно использовать флаг командной строки -w или включить в текущую лексическую область видимости прагму use warnings "untie". В том и другом случае выявляется обращение к untie, в то время как сохраняются ссылки на связанный объект. Если это так, то Perl выводит предупредительное сообщение:

```
untie attempted while 1 inner references still exist
[попытка untie при существовании 1 внутренних ссылок]
```

Чтобы заставить программу работать правильно и подавить вывод предупредительных сообщений, ликвидируйте лишние ссылки на связанный объект *перед* вызовом untie. Можно сделать это явным образом:

```
undef $x;
untie $fred;
```

Однако часто можно решить эту проблему, просто обеспечив выход переменных из области видимости в нужный момент.

Модули для связывания в CPAN

Прежде чем вдохновиться написать собственный модуль для связывания, следует проверить, не сделал ли уже это кто-нибудь. В CPAN масса модулей для связывания, и их число увеличивается с каждым днем. (Или, во всяком случае, с каждым месяцем.) В табл. 14.1 перечислены некоторые из них.

Таблица 14.1. Модули для связывания в CPAN

Модуль	Описание
GnuPG::Tie::Encrypt	Связывает интерфейс дескриптора файла и шифрование с помощью GNU Privacy Guard
IO::WrapTie	Оборачивает связанные объекты в интерфейс IO::Handle
MLDBM	Прозрачное сохранение сложных данных, не только простых строк, в файле DBM
Net::NISplusTied	Связывает хеши с таблицами NIS+
Tie::Cache::LRU	Реализует кэш для наиболее давно использованных
Tie::Const	Предоставляет константные скаляры и хеши
Tie::Counter	Заставляет скалярную переменную увеличиваться при каждом обращении
Tie::CPHash	Реализует не чувствительный к регистру, но сохраняющий его хеш
Tie::DB_FileLock	Обеспечивает блокирующий доступ к базе данных Berkeley DB 1.x
Tie::DBI	Связывает хеши с реляционными базами данных DBI
Tie::DB_Lock	Связывает хеши с базами данных с использованием совместных и монопольных блокировок
Tie::Dict	Связывает хеш с сервером RPC dict
Tie::Dir	Связывает хеш для чтения каталогов
Tie::DirHandle	Связывает указатели каталогов
Tie::FileLRUCache	Реализует облегченный, базирующийся на файловой системе, постоянный кэш LRU (наиболее давно использованных)
Tie::FlipFlop	Реализует связывание, чередующееся между двумя значениями
Tie::HashDefaults	Позволяет хешу иметь значения по умолчанию
Tie::HashHistory	Прослеживает последовательность всех изменений в хеше
Tie::IxHash	Предоставляет упорядоченные ассоциативные массивы для Perl
Tie::LDAP	Реализует интерфейс к базе данных LDAP
Tie::Persistent	Предоставляет постоянные структуры данных через tie

Таблица 14.1 (продолжение)

Модуль	Описание
Tie::Pick	Случайным образом выбирает (и удаляет) элемент множеств
Tie::RDBM	Связывает хеши с реляционными базами данных
Tie::SecureHash	Поддерживает инкапсуляцию на основе пространств имен
Tie::STDERR	Направляет выдачу с STDERR в другой процесс, например почтовую программу
Tie::Syslog	Привязывает дескриптор файла для автоматического вывода в системный журнал
Tie::TextDir	Привязывает каталог файлов
Tie::TransactHash	Редактирует хеш в транзакциях без изменения порядка во время транзакции
Tie::VecArray	Предоставляет интерфейс массива к вектору битов
Tie::Watch	Помещает контрольные точки на переменные Perl
Win32::TieRegistry	Предоставляет мощные и простые способы работы с реестром Microsoft Windows

III

Perl как технология

15

Unicode

Если вы еще не знаете, что такое Unicode, то скоро узнаете – даже если пропустите эту главу, – потому что работа с Unicode становится неизбежностью. (Некоторые рассматривают ее как неизбежное зло, но скорее это неизбежное добро. Во всяком случае это неизбежное бремя.)

Исторически, люди создавали наборы символов, отражающие их потребности в контексте собственной культуры. Поскольку во всех культурах людям свойственна естественная лень, они стремились включать только необходимые им символы и исключать ненужные. Все было прекрасно, пока мы общались только с людьми, принадлежащими нашей собственной культуре, но теперь, когда мы начинаем использовать Интернет для обмена информацией между разными культурами, такой исключающий подход служит источником проблем. Не так просто разобраться, как вводить символы с акцентами с помощью американской клавиатуры. И что на свете (в буквальном смысле) нужно, чтобы создать многоязычную веб-страницу?

Ответ или по крайней мере часть его, дает Unicode (а также XML). Unicode является включающим, а не исключающим набором символов. Хотя можно и нужно торговаться по поводу отдельных деталей работы с Unicode (а таких деталей очень много), общее желание заключается в том, чтобы все были в достаточной мере удовлетворены[1] Unicode и стремились к его использованию в качестве интернационального носителя для обмена текстовыми данными. Никто не заставляет вас применять Unicode, как никто не заставляет

---

[1] Или, в некоторых случаях, недостаточно несчастны.

вас читать эту главу (мы надеемся). Люди всегда смогут использовать старые исключающие наборы символов внутри своей собственной культуры. Однако в таком случае страдает переносимость.

Закон сохранения страдания гласит, что если уменьшить страдания в одном месте, они возрастут в некотором другом месте. В случае Unicode мы должны пострадать при переходе от байтовой семантики к символьной семантике. Поскольку по прихоти истории Perl был изобретен американцем, он исторически путает понятия байта и символа. При переходе на Unicode Perl должен каким-то образом их распутать.

Парадоксально, но, заставляя Perl самостоятельно распутывать байты и символы, мы можем позволить программирующему на Perl путать их и полагаться на то, что Perl разберется, – так же, как мы позволяем программистам путать числа и строки и полагаться на Perl в проведении необходимых преобразований. В допустимых пределах подход Perl к Unicode такой же, как ко всему остальному: Делай Правильную Вещь. В идеале хотелось бы достичь следующих четырех целей:

Цель №1:

Старые байт-ориентированные программы не должны случайно спотыкаться на старых байт-ориентированных данных, с которыми привыкли работать.

Цель №2:

Старые байт-ориентированные программы должны чудесным образом начать работать с новыми символьно-ориентированными данными, когда это потребуется.

Цель №3:

Программы должны выполняться с одинаковой скоростью как в новом символьно-ориентированном режиме, так и в старом байт-ориентированном режиме.

Цель №4:

Perl должен сохраниться как единый язык, а не разветвиться на байт-ориентированный Perl и символьно-ориентированный Perl.

В совокупности достичь этих целей практически невозможно. Но мы весьма приблизились к ним. Или, скорее, мы находимся в процессе значительного приближения, т. к. работа продолжается. По мере развития Unicode то же происходит и с Perl. Но наш всеобъемлющий план нацелен на обеспечение безопасного маршрута перехода, который приведет нас туда, куда нужно, с минимальными потерями в пути. Как мы этого добиваемся, рассказывается в следующем разделе.

Байты и символы

В версиях Perl до 5.6 все строки рассматривались как последовательности байт.[1] Однако в версиях начиная с 5.6 строка может содержать символы, которые шире одного байта. Мы теперь рассматриваем строки не как последовательности байт, а как последовательности чисел в диапазоне `0 .. 2**32-1` (или, в случае 64-разрядных компьютеров, `0 .. 2**64-1`). Эти числа представляют абстрактные символы, и чем больше число, тем, в некотором смысле, «шире» символ; но, в отличие от многих других языков, Perl не привязан к какой-либо конкретной ширине представления символов. Perl использует кодировку переменной длины (на основе UTF-8), поэтому числа для этих абстрактных символов не обязательно упаковываются по одному в байт. Очевидно, что если некоторому символу соответствует число `18 446 744 073 709 551 615` (т. е. `\x{ffff_ffff_ffff_ffff}`), то оно не поместится в одном байте (на самом деле оно занимает 13 байт), но если все символы строки находятся в диапазоне десятичных чисел `0..127`, то, конечно, они упаковываются по одному в байт, т. к. UTF-8 совпадает с ASCII в младших семи разрядах.

Perl использует UTF-8, только когда считает это выгодным, поэтому если все символы вашей строки находятся в диапазоне `0..255`, есть большая вероятность, что все символы упакованы в байты, но при отсутствии других сведений в этом нельзя быть уверенным, поскольку внутренне Perl при необходимости осуществляет преобразование между символами фиксированной 8-разрядной длины и символами переменной длины UTF-8. Суть в том, что обычно вам не приходится об этом беспокоиться, т. к. на абстрактном уровне семантика символов сохраняется независимо от представления.

В любом случае, если в строке есть символы, числовое значение которых больше `255` в десятичной системе, строка обязательно хранится в UTF-8. Точнее, она хранится в расширенной Perl-версии UTF-8, которую мы называем *utf8* в честь прагмы с таким именем, но главным образом потому, что это легче вводить. (И потому, что в «настоящей» UTF-8 допускаются только численные значения символов, утвержденные Консорциумом Unicode. В utf8 разрешается иметь любые численные значения символов, которые вам нужны для работы. Perl не волнует, являются ли значения ваших символов официально корректными или просто корректными.)

Мы сказали, что по большей части вам не приходится об этом беспокоиться, но людям все равно хочется беспокоиться. Допустим, вы используете v-строку для представления адреса IPv4:

```
$locaddr = v127.0.0.1;      # Наверняка хранится как байты.
$oreilly = v204.148.40.9;   # Может храниться как байты или utf8.
$badaddr = v2004.148.40.9;  # Наверняка хранится как utf8.
```

[1] Можете называть их «октетами», но мы считаем, что в наше время эти слова стали почти синонимами, поэтому будем придерживаться слова, используемого «синими воротничками».

Каждый может сообразить, что эта `$badaddr` не будет действовать в качестве IP-адреса. Поэтому можно подумать, что если сетевой адрес O'Reilly's перевести в представление UTF-8, он больше не сможет работать. Но символы в строке – это абстрактные числа, а не байты. Всюду, где используются адреса IPv4, например в функции `gethostbyaddr`, числа абстрактных символов должны автоматически принудительно преобразовываться обратно в байтовое представление (а для `$badaddr` должна возникать ошибка).

Интерфейсы между Perl и реальным миром должны учитывать детали представления. В допустимых пределах существующие интерфейсы стараются поступать правильно без дополнительных указаний. Но иногда приходится давать инструкции некоторым интерфейсам, например функции `open`), а если вы пишете собственный интерфейс к реальному миру, он должен быть настолько интеллектуален, чтобы разбираться в обстановке самостоятельно, или хотя бы достаточно интеллектуален, чтобы выполнять команды, задающие режим работы, отличный от установленного по умолчанию.[1]

Поскольку Perl старается поддерживать прозрачную семантику символов внутри самого языка, единственное место, где программиста должна беспокоить символьная и байтовая семантика, – это его интерфейсы. По умолчанию все старые интерфейсы Perl с окружающим миром являются байт-ориентированными, поэтому они создают и принимают байт-ориентированные данные. Это значит, что на абстрактном уровне все строки являются последовательностями чисел в диапазоне `0..255`, поэтому, если ваша программа не переводит их принудительно в представление utf8, то ваша прежняя программа продолжает работать с байт-ориентированными данными, как и ранее. Так что можете пометить галочкой указанную выше Цель №1.

Если вы хотите, чтобы старая программа работала с новыми символьно-ориентированными данными, пометьте свои символьно-ориентированные интерфейсы, чтобы Perl рассчитывал на получение символьно-ориентированных данных от этих интерфейсов. После того как вы это сделаете, Perl должен автоматически произвести преобразования, необходимые для сохранения символьной абстракции. Единственное отличие в том, что в программу вводятся строки, помеченные как потенциально содержащие символы, б*о*льшие чем `255`, поэтому при выполнении операции над строкой байт и строкой utf8 Perl внутренне преобразует байтовую строку в строку utf8, прежде чем выполнить операцию. Обычно строки utf8 преобразуются обратно в байтовые, только если вы посылаете их на байтовый интерфейс, и тогда, если строка содержит символы, б*о*льшие чем `255`, возникает проблема, кото-

[1] В некоторых системах могут присутствовать средства для одновременного переключения всех ваших интерфейсов. Если указывается ключ командной строки `-C` (или глобальная переменная `${^WIDE_SYSTEM_CALLS}` установлена равной 1), то все системные вызовы будут использовать соответствующие API расширенных символов. (В настоящее время это реализовано только в Microsoft Windows.) В текущие планы сообщества Linux входит, чтобы все интерфейсы переключались в режим UTF-8, когда `$ENV{LC_CTYPE}` имеет значение «`UTF-8`». В других сообществах может быть принят другой подход. Пройденные нами пути могут отличаться.

рую можно решать разными способами в зависимости от конкретного интерфейса. Так что можете пометить галочкой указанную выше Цель №2.

Иногда требуется смешать код, понимающий символьную семантику, с кодом, который должен выполняться с байтовой семантикой, например в коде ввода/вывода, который читает или пишет блоки фиксированного размера. В таком случае можно поместить объявление `use bytes` вокруг байт-ориентированного кода и заставить его использовать байтовую семантику даже со строками, помеченными как строки utf8. Ответственность за все необходимые преобразования лежит в этом случае на вас. Но это способ более строгого локального прочтения Цели №1 за счет более свободного глобального прочтения Цели №2.

Цель №3 в значительной мере достигнута, частично путем откладываемых преобразований между представлениями в байтах и utf8, а частично путем увиливания от реализации потенциально замедляющих функций Unicode, таких как поиск свойств символов в гигантских таблицах.

Цель №4 достигнута в результате пожертвования небольшой частью совместимости интерфейсов ради достижения остальных Целей. Можно считать, что мы не разветвили Perl надвое, но можно посмотреть и так, что версия Perl 5.6 *является* ответившейся версией Perl по отношению к более ранним версиям, и мы не думаем, что люди станут переходить на нее с более ранних версий, прежде чем убедятся, что новая версия делает то, что им нужно. Но так всегда бывает с новыми версиями, поэтому позволим себе также поставить галочку и у Цели №4.

Действие символьной семантики

Вывод из всего этого такой, что обычный встроенный оператор будет действовать над символами, если только не окажется в области действия прагмы `use bytes`. Однако даже вне области действия `use bytes`, если все операнды оператора хранятся как 8-разрядные символы (т. е. ни один операнд не хранится в utf8), символьная семантика неотличима от байтовой, а результат оператора внутренне хранится в 8-разрядном формате. В результате сохраняется обратная совместимость, если только не подавать в программу символы, находящиеся за пределами Latin-1.

Прагма `utf8` является, по преимуществу, средством обеспечения совместимости, которое активирует распознавание UTF-8 в литералах и идентификаторах, встречающихся синтаксическому анализатору. Она может также применяться для активизации некоторых более экспериментальных функций поддержки Unicode. Наша долгосрочная цель состоит в превращении прагмы `utf8` в код пустой операции.

Прагма `use bytes` никогда не станет кодом пустой операции. Она не только необходима для байт-ориентированного кода, но и оказывает побочный эффект, определяя байт-ориентированные оболочки для некоторых функций, применяемые вне области действия `use bytes`. На момент написания

данной книги единственная такая оболочка определена для `length`, но со временем, вероятно, появятся другие. Для использования такой оболочки нужно сказать:

```
use bytes ();   # Загрузить оболочки без импорта байтовой семантики.
...
$charlen =        length("\x{ffff_ffff}");   # Возвращает 1.
$bytelen = bytes::length("\x{ffff_ffff}");   # Возвращает 7.
```

Вне области видимости объявления `use bytes` Perl версии 5.6 работает (или, по крайней мере, должен работать) следующим образом:

- В строках и шаблонах теперь могут содержаться символы с порядковым значением больше 255:

  ```
  use utf8;
  $convergence = "☞ ☜";
  ```

  В предположении, что вы пишете программу в редакторе с поддержкой Unicode, такие символы обычно встречаются непосредственно в литеральных строках в виде символов UTF-8. В настоящее время для того, чтобы иметь возможность использовать UTF-8 в литералах, необходимо объявить `use utf8` в начале программы.

  Если редактора с поддержкой Unicode нет, всегда можно задать конкретный символ в ASCII, используя расширение в виде нотации \x. Символ в диапазоне Latin-1 можно записать в виде \x{ab} или \xab, но если число содержит больше двух шестнадцатеричных цифр, то его следует заключить в фигурные скобки. Символы Unicode задаются с помощью шестнадцатеричного кода в фигурных скобках после \x. Например, смайлик ☺ в Unicode обозначается как \x{263A}. В Perl нет синтаксических конструкций, предполагающих, что символы Unicode состоят ровно из 16 бит, поэтому нельзя, как в других языках, написать \u263A; ближайшим эквивалентом служит \x{263A}.

  Как вставлять именованные символы с помощью `\N{CHARNAME}`, см. в описании прагмы `use charnames` в главе 31 «Модули прагм».

- Идентификаторы в сценарии Perl могут содержать буквенно-цифровые символы Unicode, в том числе идеографические символы (идеограммы):

  ```
  use utf8;
  $人++;        # Ребенок родился.[1]
  ```

  И снова для распознавания UTF-8 в сценарии требуется (пока) `use utf8`. В данное время вы можете самостоятельно принимать решение об использовании канонических форм символов – Perl не пытается (пока) канонизировать имена переменных вместо вас. Мы рекомендуем применять в программах канонизацию согласно нормализованной форме C, посколь-

[1] Иероглиф 人 означает «человек». – *Примеч. науч. ред.*

ку когда-нибудь Perl будет делать это по умолчанию. См. на *www.unicode.org* последний технический доклад по канонизации.

- Регулярные выражения ищут символы, а не байты. К примеру, точка соответствует символу, а не байту. Если Консорциум Unicode когда-нибудь одобрит алфавит Tengwar, то (несмотря на то, что такие символы представляются четырьмя байтами UTF-8) он будет соответствовать поиску:

```
"\N{TENGWAR LETTER SILME NUQUERNA}" =~ /^.$/
```
[1]

  Шаблон \C служит для обеспечения поиска одного байта («char» в C, отсюда и \C). Применяйте \C с осторожностью, поскольку он может вызвать рассинхронизацию с границами символов в строке и будет получена ошибка «Malformed UTF-8 character» (неправильный символ UTF-8). Нельзя использовать \C в квадратных скобках, поскольку он не представляет какой-либо конкретный символ или группу символов.

  Классы символов в регулярных выражениях сравниваются с символами, а не байтами, и сравниваются со свойствами символов, заданными в базе данных свойств Unicode. Поэтому \w может применяться для поиска идеографического символа (идеограммы):

```
"人" =~ /\w/  </para>
```

- Можно использовать именованные свойства и диапазоны блоков Unicode как классы символов посредством новых конструкций \p (соответствует свойству) и \P (не соответствует свойству). Например, \p{Lu} соответствует любому символу в верхнем регистре в смысле Unicode, а \p{M} соответствует любому символу маркировки. Для однобуквенных свойств скобки могут быть опущены, поэтому символы маркировки можно искать также с помощью \pM. Есть много предопределенных классов символов, например \p{IsMirrored} и \p{InTibetan}:

```
"\N{greek:Iota}" =~ /\p{Lu}/
```

  Можно также использовать \p и \P в квадратных скобках классов символов. (В версии Perl 5.6.0 требуется объявить use utf8, чтобы свойства символов правильно работали. В дальнейшем это ограничение будет снято.) О подробностях, связанных с поиском свойств Unicode, см. главу 5 «Поиск по шаблону».

- Особый шаблон \X ищет расширенные последовательности Unicode (на языке Стандарта – «комбинированная символьная последовательность» (combining character sequence)), в которых первый символ является базовым, а последующие символы – маркерами, примененными к базовому символу. Это эквивалентно (?:\PM\pM*):

```
"o\N{COMBINING TILDE BELOW}" =~ /\X/
```

---

[1] Tengwar letter silme nuquerna – буква («тенгва») «сильме перевернутая» ᘐ из алфавита Tengwar. – *Примеч. ред.*

Не разрешается использовать \X в квадратных скобках, поскольку он может найти несколько символов и не соответствует какому-либо конкретному символу или набору символов.

- Оператор tr/// транслитерирует символы вместо байтов. Чтобы преобразовать все символы вне диапазона Latin-1 в вопросительный знак, можно сказать:

```
tr/\0-\x{10ffff}/\0-\xff?/;       # utf8 в символы latin1
```

- Операторы перевода регистра, получая на входе символы, обращаются к таблицам перевода регистра Unicode. Обратите внимание, что uc переводит в верхний регистр, а ucfirst – в заглавный регистр (для языков, в которых такое различие есть). Естественно, что соответствующие последовательности с обратной косой чертой имеют ту же семантику:

```
$x = "\u$word";       # перевести в заглавный регистр первую букву $word
$x = "\U$word";       # перевести в верхний регистр $word
$x = "\l$word";       # перевести в нижний регистр первую букву $word
$x = "\L$word";       # перевести в нижний регистр  $word
```

Будьте осторожны, поскольку эти таблицы не пытаются обеспечить полное соответствие прямого и обратного преобразований для каждого случая, особенно в языках, использующих различное число символов в заглавном или верхнем регистрах и эквивалентной букве нижнего регистра. Как сказано в стандарте, хотя сами свойства регистра являются нормативными, соответствия регистров несут только информационную нагрузку.

- Большинство операторов, работающих с позицией в строке или ее длиной, автоматически переходят на использование позиций символов, в том числе chop, substr, pos, index, rindex, sprintf, write и length. В число операторов, которые умышленно не переключаются, входят vec, pack и unpack. В число операторов, которым это безразлично, входят chomp, а также все операторы, рассматривающие строку как битовую корзину, например, sort по умолчанию и операторы, обрабатывающие имена файлов.

```
use bytes;
$bytelen = length("I do &合氣道.");   # 15 байт
no bytes;
$charlen = length("I do 合氣道.");   # но 9 символов
```

- Буквы "c" и "C" в pack/unpack *не* меняются, поскольку часто используются для байт-ориентированных форматов. (И снова представьте себе "char" в языке C.) Однако есть новый спецификатор "U", осуществляющий преобразование между символами UTF-8 и целыми числами:

```
pack("U*", 1, 20, 300, 4000) eq v1.20.300.4000
```

- Функции chr и ord работают с символами:

```
chr(1).chr(20).chr(300).chr(4000) eq v1.20.300.4000
```

Иными словами, chr и ord действуют как pack("U") и unpack("U"), а не как pack("C") и unpack("C"). На самом деле, две последние осуществляют теперь эмуляцию байт-ориентированных chr и ord, если вам слишком лень написать use bytes.

- И наконец scalar reverse обращает посимвольно, а не побайтно:

```
"☞ ✍" eq reverse "✍ ☞" </para>
```

Если посмотреть в каталог *ПУТЬ_К_PERLLIB/unicode*, то можно найти ряд файлов, имеющих отношение к определению вышеописанной семантики. База данных свойств Unicode Консорциума Unicode находится в файле с именем *Unicode.300* (для Unicode 3.0). Этот файл уже преобразован *mktables.PL* в кучу маленьких файлов *.pl* в том же самом каталоге (и подкаталогах *Is/*, *In/* и *To/*), некоторые из которых Perl автоматически проглатывает, чтобы реализовывать такие вещи, как \p (см. каталоги *Is/* и *In/*) и uc (см. каталог *To/*). Другие файлы заглатываются такими модулями, как прагма use charnames (см. *Name.pl*). Но во время написания этой книги там было еще много файлов, ожидавших, когда вы создадите модуль для доступа к ним:

ArabLink.pl
ArabLnkGrp.pl
Bidirectional.pl
Block.pl
Category.pl
CombiningClass.pl
Decomposition.pl
JamoShort.pl
Number.pl
To/Digit.pl

Значительно более удобочитаемая сводка по Unicode с многочисленными ссылками находится в *ПУТЬ_К_PERLLIB/unicode/Unicode3.html*.

Заметьте, что когда консорциум Unicode выпускает новую версию, некоторые из этих имен могут изменяться, поэтому следует быть осмотрительным. Найти *ПУТЬ_К_PERLLIB* можно с помощью следующего заклинания:

```
% perl -MConfig -le 'print $Config{privlib}'
```

Практически обо всем, что относится к Unicode, можно узнать, выписав *The Unicode Standard, Version 3.0* (ISBN 0-201-61633-5).

Осторожно, работают 人

Во время написания этой книги (т. е. речь идет о версии Perl 5.6.0) сохранялся ряд предостережений по использованию Unicode. (Проверьте возможные обновления в своей электронной документации.)

- Существующий компилятор регулярных выражений не создает полиморфных кодов операций. Это означает, что некоторый шаблон определяется как способный искать символы Unicode во время его компиляции (на основании присутствия в шаблоне символов Unicode), а не во время поиска на этапе исполнения. Это нужно изменить, чтобы настраиваться на поиск Unicode, когда строка для поиска представлена в Unicode.
- В настоящее время нет простого способа пометки данных, читаемых из файла или другого внешнего источника, как utf8. На этом в ближайшем будущем будут сосредоточены большие усилия, и положение, возможно, уже изменится, когда вы будете читать эту книгу.
- Нет метода для автоматического перевода входных или выходных данных в кодировку, отличную от UTF-8. Однако это планируется сделать в ближайшее время, поэтому сверьтесь со своей электронной документацией.
- Применение национальных установок с utf8 может привести к неожиданным результатам. В настоящее время делаются некоторые попытки использования 8-разрядной информации о национальных настройках с символами в диапазоне `0..255`, но оно с очевидностью некорректно для национальных установок, затрагивающих символы за пределами этого диапазона (при отображении в Unicode). При этом также происходит замедление работы. Настоятельно рекомендуется избегать национальных установок.

Unicode – это забавно, нужно только корректно определить, что такое забавно.

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 5-93286-020-0, название «Программирование на Perl, 3-е издание» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.

16

Межпроцессное взаимодействие

У процессов в компьютере почти столько же способов для общения друг с другом, сколько у людей. Сложность взаимодействия между процессами не следует недооценивать. Бесполезно искать смысл в словах, если ваш приятель говорит на языке жестов. Аналогично, два процесса могут общаться между собой, только договорившись о средствах связи и основываясь на соглашениях, построенных поверх них. Как и при любом обмене информацией, соглашения, о которых нужно договориться, простираются от лексических до практических: от используемого жаргона до очередности вступления в разговор. Эти соглашения необходимы, потому что очень трудно передавать голую семантику в отсутствие контекста.

На нашем жаргоне межпроцессное взаимодействие обычно называется IPC (Interprocess Communication). В Perl представлены средства взаимодействия IPC от самых простых до самых сложных. Выбор средства зависит от сложности передаваемой информации. В простейших случаях почти никакая информация не передается: просто поступает уведомление о том, что некоторое событие произошло в некоторый момент. В Perl о таких событиях сообщается с помощью сигнального механизма, сделанного по образцу системы сигналов Unix.

С другой стороны, в Perl находятся средства поддержки сокетов, позволяющие обмениваться информацией с любым другим процессом в Интернете в соответствии с выбранным взаимно поддерживаемым протоколом. Конечно, за такую свободу приходится расплачиваться: нужно пройти через ряд этапов, чтобы установить соединения и обеспечить использование языка, понятного другому процессу. Это, в свою очередь, может потребовать соблюде-

ния некоторого количества других странных обычаев, зависящих от местных нравов. Для выполнения всех требований протокола вас могут даже обязать разговаривать на одном из таких языков, как XML, Java или Perl. Ужас.

Посередине располагаются средства, предназначенные, в основном, для связи с процессами на той же машине. В их число входят старые добрые файлы, каналы, очереди FIFO и различные системные вызовы System V IPC. Поддержка этих средств зависит от платформы; современные Unix-системы (в том числе Apple Mac OS X) должны поддерживать все средства, а большинство средств, за исключением сигналов и SysV IPC, поддерживается в новых операционных системах Microsoft, включая каналы, ветвление, блокировку файлов и сокеты.[1]

Дополнительные сведения о переносимости в целом можно найти в стандартном наборе документации Perl (в любом формате, поддерживаемом вашей системой) через *perlport*. Специфическую для Microsoft информацию можно найти через *perlwin32* и *perlfork*, которые устанавливаются даже на системах, отличающихся от Microsoft. В качестве учебников предлагаем следующие:

- «Perl Cookbook», Tom Christiansen и Nathan Torkington, O'Reilly & Associates, 1998 («Perl: библиотека программиста», Том Кристиансен и Натан Торкингтон, изд-во «Питер», 2000).
- «Advanced Programming in the UNIX Environment», W. Richard Stevens (Addison-Wesley, 1992).
- «TCP/IP Illustrated», W. Richard Stevens, тома I–III (Addison-Wesley, 1992–1996).

Сигналы

В Perl применяется простая модель обработки сигналов: хеш `%SIG` содержит ссылки (символические или жесткие) на определенные пользователем обработчики сигналов. Некоторые события побуждают операционную систему доставлять сигнал затронутому ими процессу. Обработчик, соответствующий этому событию, вызывается с передачей ему одного аргумента, содержащего название запустившего его сигнала. Для отправки сигнала другому процессу используется функция `kill`. Можно считать это передачей процессу одного бита информации.[2] Если в этом процессе для данного сигнала установлен обработчик, то при получении сигнала он может выполнить некоторый код. Однако передающему процессу не может быть возвращено никакого ответного значения, кроме известия о том, что сигнал был успешно отправлен. Отправитель не имеет никакой обратной связи, чтобы узнать, что сделал с сигналом принимающий процесс, и сделал ли вообще.

---

[1] За исключением сокетов `AF_UNIX`.

[2] На самом деле это, скорее всего, пять или шесть битов в зависимости от того, сколько сигналов определяет операционная система, и от того, используют ли другие процессы тот факт, что вы *не* послали другой сигнал.

Мы классифицировали эту возможность как форму IPC, но на самом деле сигналы могут поступать из разных источников, не только от других процессов. Сигнал может поступить и от вашего собственного процесса или быть сгенерирован, когда пользователь вводит с клавиатуры особую последовательность, например <Ctrl>+<C> или <Ctrl>+<Z>, или производиться ядром, когда происходит некоторое событие, например, завершение работы дочернего процесса, исчерпание памяти стека вашим процессом, превышение границ размера файла или памяти. Однако вашему собственному процессу нелегко различить эти случаи. Сигнал – это как посылка, таинственным образом оказавшаяся на вашем пороге и не имеющая обратного адреса. Вскрывать ее нужно с осторожностью.

Поскольку элементами массива `%SIG` могут быть жесткие ссылки, общепринято использовать в качестве обработчиков простых сигналов анонимные функции:

```
$SIG{INT}  = sub { die "\nOutta here!\n" };
$SIG{ALRM} = sub { die "Your alarm clock went off" };
```

Можно также создать именованную функцию и присвоить ее имя или ссылку на нее соответствующей позиции хеша. Например, для перехвата сигналов прерывания и завершения (часто привязываемых к <Ctrl>+<C> и <Ctrl>+<\> на клавиатуре) установите такой обработчик:

```
sub catch_zap {
    my $signame = shift;
    our $shucks++;
    die "Кто-то послал мне SIG$signame!";
}
$shucks = 0;
$SIG{INT} = 'catch_zap';   # всегда означает &main::catch_zap
$SIG{INT} = \&catch_zap;   # самый лучший вариант
$SIG{QUIT} = \&catch_zap;  # перехват еще одного сигнала
```

Обратите внимание, что в обработчике сигналов мы лишь устанавливаем глобальную переменную и инициализируем исключительную ситуацию посредством `die`. Старайтесь по возможности избегать чего-либо более сложного, потому что в большинстве систем библиотека C не реентерабельна. Передача сигналов происходит асинхронно,[1] поэтому вызов любых функций `print` (или даже чего-то, требующего выделения дополнительной памяти через *malloc* (3)) теоретически может вызвать ошибку распределения памяти с последующим выводом дампа памяти, если в момент передачи сигнала вы уже находились в соответствующей подпрограмме библиотеки C. (Даже программа `die` не вполне безопасна, если только процесс не выполняется в рамках

[1] Синхронизация передачи сигналов с кодами операций уровня Perl запланирована для включения в будущие выпуски Perl, что должно решить проблему сигналов и дампов памяти.

eval, которая подавляет ввод/вывод от die, благодаря чему предотвращается обращение к библиотеке C. Вероятно.)

Еще более простой способ перехвата сигналов состоит в использовании прагмы sigtrap для установки простых обработчиков сигналов по умолчанию:

```
use sigtrap qw(die INT QUIT);
use sigtrap qw(die untrapped normal-signals
   stack-trace any error-signals);
```

Эта прагма полезна, если вы не хотите утруждать себя созданием собственного обработчика, но все же хотите перехватывать опасные сигналы и осуществлять контроль над завершением работы. По умолчанию некоторые из этих сигналов могут быть настолько смертельными для вашего процесса, что ваша программа просто замрет на месте, получив один из них. К несчастью, это означает, что все функции END для обработки при завершении и методы DESTROY для закрытия объектов не будут вызваны. Однако они вызываются при обычных исключительных ситуациях Perl (например, при вызове die), поэтому данную прагму можно использовать для безболезненного преобразования сигналов в исключительные ситуации. Даже несмотря на то, что сами вы не обрабатываете сигналы, ваша программа ведет себя корректно. Читайте в описании use sigtrap в главе 31 «Модули прагм» о многих дополнительных возможностях этой прагмы.

Можно также установить обработчик в %SIG в одну из строк "IGNORE" или "DEFAULT", и тогда Perl попытается отбросить сигнал или разрешить для него действие по умолчанию (хотя некоторые сигналы нельзя ни перехватить, ни игнорировать, например сигналы KILL и STOP; смотрите в *signal*(3), если он у вас есть, список сигналов, имеющихся в системе, и их режимы по умолчанию).

В операционной системе сигналы представлены как числа, а не как имена, но Perl, как большинство людей, предпочитает символические имена магическим числам. Чтобы узнать имена сигналов, выведите ключи хеша %SIG или выполните команду *kill -l*, если она поддерживается системой. Можно также использовать стандартный модуль Perl Config, чтобы определить соответствие в операционной системе между именами сигналов и номерами сигналов. Пример см. в *Config*(3).

Поскольку %SIG представляет собой глобальный хеш, то изменения в нем воздействуют на всю программу. Часто более тактично по отношению к остальной части программы заключить свою обработку сигналов в ограниченную область видимости. Сделайте это при помощи обработчика сигнала local, который прекратит свое действие после выхода из охватывающего блока. (Но помните, что значения local видны в функциях, вызываемых изнутри этого блока.)

```
{
    local $SIG{INT} = 'IGNORE';
    ...     # Делайте здесь, что хотите, игнорируя все SIGINT.
    fn();   # SIGINT игнорируются внутри fn() тоже!
    ...     # И здесь.
```

```
}           # Выход из блока восстанавливает прежнее значение $SIG{INT}.
fn();       # SIGINT не игнорируются внутри fn() (предположительно).
```

Передача сигналов группе процессов

Процессы (по крайней мере, в Unix) организуются в группы, обычно соответствующие заданию в целом. Например, при запуске одной команды оболочки, состоящей из последовательности команд-фильтров, передающих данные по конвейеру одна другой, все эти процессы (и их дочерние процессы) принадлежат одной и той же группе процессов. Эта группа процессов имеет номер, соответствующий номеру процесса лидера группы. Если, посылая сигнал, задать положительный номер процесса, то сигнал будет просто отправлен указанному процессу, но если задать отрицательный номер, то сигнал отправляется каждому процессу, номер группы процессов которого равен соответствующему положительному числу, т. е. номеру процесса лидера группы процессов. (К удобству для лидера группы процессов, ID группы процессов – это просто `$$`.)

Предположим, что программа хочет послать сигнал отбоя (hang-up) всем дочерним процессам, которые она запустила непосредственно, плюс внукам, запущенным этими дочерними процессами, плюс правнукам, запущенным этими внуками, и т. д. Для этого программа сначала вызывает `setpgrp(0,0)`, чтобы стать лидером новой группы процессов, и все процессы, которые они создают, будут входить в новую группу. Не важно, как будут эти процессы запускаться: вручную через `fork`, автоматически через конвейеризованные `open` или как фоновое задание посредством `system("cmd &")`. Даже если у тех процессов есть собственные потомки, отправка сигнала отбоя группе процессов в целом найдет их все (за исключением процессов, которые организовали собственную группу или поменяли свой UID, чтобы получить «дипломатический иммунитет» от ваших сигналов).

```
{
    local $SIG{HUP} = 'IGNORE';   # исключить себя самого
    kill(HUP, -$$);               # послать сигнал своей
                                  # собственной группе процессов
}
```

Еще одним интересным сигналом является сигнал с номером `0`. Фактически он не оказывает влияния на целевой процесс, а проверяет, жив ли тот и не поменял ли свой UID. Таким образом, он проверяет, законна ли посылка сигнала, но фактически не отправляет его.

```
unless (kill 0 => $kid_pid) {
    warn "что-то нехорошее произошло с $kid_pid";
}
```

Сигнал с номером `0` является единственным сигналом, который работает одинаково в версиях Perl под Microsoft и Unix. В системах Microsoft `kill` фактически не передает сигнал. Вместо этого он заставляет целевой процесс

выйти со статусом, заданным номером сигнала. Когда-нибудь это может быть исправлено. Однако волшебный сигнал 0 работает стандартным неразрушающим образом.

Уборка зомби

По завершении процесса ядро посылает его родителю сигнал CHLD, и процесс превращается в зомби,[1] пока его родитель не вызовет wait или waitpid. Если запустить в Perl другой процесс, не используя для этого fork, то Perl берет заботу об уборке зомбированных потомков на себя, но в случае применения голого fork предполагается, что вы сами убираете за собой. Во многих, но не во всех ядрах простым приемом для автоматической уборки зомби является установка $SIG{CHLD} в 'IGNORE'. Более гибкий (но трудоемкий) подход состоит в самостоятельной их уборке. В связи с тем, что несколько потомков могут умереть, прежде чем вы соберетесь заняться ими, необходимо собирать своих зомби в цикле, пока их больше не останется:

```
use POSIX ":sys_wait_h";
sub REAPER { 1 until waitpid(-1, WNOHANG) == -1) }
```

Чтобы запускать этот код должным образом, можно установить обработчик сигнала CHLD:

```
$SIG{CHLD} = \&REAPER;
```

или, если программа работает в цикле, сделать так, чтобы «уборщик» запускался достаточно часто. Это самый лучший подход, поскольку он не связан со случайными дампами памяти, которые иногда запускают сигналы в библиотеке C. Однако он дорого обходится, если постоянно вызывать его в цикле, поэтому разумным компромиссом является гибридная стратегия, при которой программист минимизирует риск в обработчике, делая в нем как можно меньше и убирая зомби, только оказавшись снаружи:

```
our $zombies = 0;
$SIG{CHLD} = sub { $zombies++ };
sub reaper {
    my $zombie;
    our %Kid_Status;  # запомнить каждый статус выхода
    $zombies = 0;
    while (($zombie = waitpid(-1, WNOHANG)) != -1) {
        $Kid_Status{$zombie} = $?;
    }
}
while (1) {
    reaper() if $zombies;
    ...
}
```

[1] Это действительно технический термин.

В данном коде предполагается, что ядро поддерживает надежные сигналы. Старое ядро SysV обычно этого не делало, из-за чего написание для него надежных обработчиков сигналов было невозможным. Начиная с версии 5.003 Perl использует, если это возможно, системный вызов *sigaction*(2), который значительно более надежен. Это означает, что если только вы не работаете в устаревшей операционной системе или со старой версией Perl, то вам не придется переустанавливать свои обработчики и подвергаться риску пропустить сигналы. К счастью, все системы типа BSD (в том числе Linux, Solaris и Mac OS X), а также все POSIX-совместимые системы поддерживают надежные сигналы, поэтому ненадежная работа старой системы SysV относится скорее к истории, чем к сегодняшним проблемам.

В этих более новых операционных системах улучшены и многие другие вещи. Например, «медленные» системные вызовы (способные вызывать блокировку, например `read`, `wait` и `accept`) автоматически перезапускаются при прерывании сигналом. В прежние недобрые времена пользователь должен был не забывать явно проверять, не заканчивался ли каждый медленный системный вызов неудачей и установкой `$!` (`$ERRNO`) в `EINTR` и, если это так, перезапускать его. Это могло происходить не только от сигналов `INT`; даже невинные сигналы вроде `TSTP` (от <Ctrl>+<Z>) или `CONT` (при повышении приоритета задачи) прерывали системный вызов. Сейчас Perl автоматически перезапускает системный вызов, если операционная система позволяет это сделать. Обычно это рассматривается как характерная особенность (feature).

Можно узнать, поддерживается ли более строгий режим сигналов в стиле POSIX, загрузив модуль `Config` и проверив, имеет ли `$Config{d_sigaction}` истинное значение. Чтобы узнать, являются ли медленные системные вызовы перезапускаемыми, сверьтесь с системной документацией в *sigaction*(2) или *sigvec*(3) либо поищите в файле C *sys/signal.h* символы `SV_INTERRUPT` или `SA_RESTART`. Если один или оба найдены, то, вероятно, системные вызовы перезапускаемы.

Завершение медленных операций по тайм-ауту

Обычным применением сигналов является установление временных границ для длительных операций. Работая в Unix-системе (или любой другой POSIX-совместимой системе, поддерживающей сигнал `ALRM`), можно попросить ядро послать процессу в некоторый момент сигнал `ALRM`:

```
use Fcntl ':flock';
eval {
    local $SIG{ALRM} = sub { die "перезапуск будильника" };
    alarm 10;               # подать сигнал через 10 секунд
    eval {
        flock(FH, LOCK_EX)  # блокирующий монопольный lock
            or die "can't flock: $!";
    };
    alarm 0;                # отменить сигнал
};
```

```
alarm 0;                # защита от ситуации гонок
die if $@ && $@ !~ /alarm clock restart/; # повторное возбуждение
```

Если аварийный сигнал поступит во время ожидания снятия блокировки, и вы просто перехватите сигнал и выполните возврат, то попадете обратно во `flock`, потому что Perl автоматически перезапускает системные вызовы, когда может это сделать. Единственный выход состоит в том, чтобы возбудить исключительную ситуацию через `die`, а затем позволить `eval` перехватить ее. (Это срабатывает, поскольку исключительная ситуация приводит к вызову функции *long jmp*(3) из библиотеки C, который действительно выводит нас из перезапускаемого системного вызова.)

Вложенная ловушка для исключительной ситуации организована в связи с тем, что вызов `flock` инициализирует исключительную ситуацию, если на данной платформе эта функция не реализована, а требуется обеспечить сброс аварийного сигнала в любом случае. Второй `alarm 0` дается на случай, если сигнал приходит после выполнения `flock`, но до того как достигнут первый `alarm 0`. Без второго `alarm` вы подвергаетесь риску «маленькой» ситуации гонок, однако размер не играет роли: ситуация либо существует, либо нет. Мы предпочитаем, чтобы она не существовала.

Блокировка сигналов

Иногда желательно отложить прием сигнала на время выполнения критической секции кода. Просто проигнорировать сигнал вы не хотите, но то, что вы делаете, слишком важно, чтобы это можно было прервать. В хеше Perl `%SIG` блокировка сигналов не реализована, но она осуществляется в модуле `POSIX` через интерфейс к системному вызову *sigprocmask*(2):

```
use POSIX qw(:signal_h);
$sigset   = POSIX::SigSet->new;
$blockset = POSIX::SigSet->new(SIGINT, SIGQUIT, SIGCHLD);
sigprocmask(SIG_BLOCK, $blockset, $sigset)
    or die "Невозможно заблокировать сигналы INT,QUIT,CHLD: $!\n";
```

Пока эти три сигнала заблокированы, можете делать что угодно, не опасаясь, что вас потревожат. Завершив критическую секцию, разблокируйте сигналы, восстановив прежнюю маску сигналов:

```
sigprocmask(SIG_SETMASK, $sigset)
    or die "Невозможно восстановить сигналы INT,QUIT,CHLD: $!\n";
```

Если за время блокировки поступили какие-либо из этих трех сигналов, они немедленно доставляются. Если не обработаны два или более разных сигналов, они поступают в произвольном порядке. Кроме того, нельзя определить, какой сигнал сколько раз был получен за время блокировки.[1] Напри-

[1] Обычно. Подсчет сигналов может быть реализован в некоторых системах реального времени согласно последним спецификациям, но мы таких пока не встречали.

мер, если девять дочерних процессов завершились, пока вы блокировали сигналы CHLD, то ваш обработчик (если он есть) после разблокирования будет вызван только один раз. Вот поэтому при уборке зомби нужно повторять цикл, пока их совсем не останется.

Файлы

Возможно, вы никогда не рассматривали раньше файлы как механизм IPC, но на них приходится львиная доля межпроцессного взаимодействия, значительно большая, чем для всех остальных средств вместе взятых. Когда один процесс помещает свои драгоценные данные в файл, а другой процесс извлекает эти данные, между этими процессами происходит обмен информацией. Файлы представляют нечто уникальное среди всех видов IPC, о которых здесь рассказывается: как свиток папируса, погребенный в пустыне и раскопанный через тысячелетия, файл может быть поднят на свет и прочтен через длительное время после кончины его автора.[1] С учетом постоянства и относительной простоты применения популярность файлов не удивительна.

Использование файлов для передачи информации из далекого прошлого в неизведанное будущее преподносит мало сюрпризов. Файл записывается на какой-то постоянный носитель, например диск, – и все. (Если он содержит HTML, можно сообщить веб-серверу, где его искать.) Интересная задача возникает, если все стороны еще живы и пытаются общаться между собой. В отсутствие некоторого соглашения относительно того, чья очередь сказать свое слово, надежная связь невозможна; соглашения можно достичь с помощью блокировки файла, о чем рассказывается в следующем разделе. А в разделе после него («Передача указателей файлов») мы обсуждаем особые отношения, существующие между родительским процессом и его потомками и позволяющие связанным родством сторонам обмениваться информацией посредством наследования доступа к одним и тем же файлам.

Конечно, возможности файлов ограничены, если речь идет о таких вещах как удаленный доступ, синхронизация, надежность и управление сеансом. В других разделах данной главы рассказывается о различных механизмах IPC, созданных для преодоления таких ограничений.

Блокировка файлов

В многозадачной среде следует проявлять осторожность, чтобы не вступить в конфликт с другими процессами, которые пытаются обратиться к используемому вами файлу. Если все процессы осуществляют только чтение, то проблем не возникает, но как только хотя бы одному процессу понадобится что-нибудь записать в файл, возникает полный хаос, если только какой-нибудь механизм блокировки не выступает в качестве регулировщика движения.

---

[1] Если предположить, что у процесса может быть кончина.

Никогда не используйте проверку существования файла (т. е. `-e $file`) в качестве индикатора блокировки, потому что условия гонок имеют место между проверкой существования файла с данным именем и операцией, которую вы собираетесь с ним совершить (например, создать его, открыть или уничтожить). Дополнительно об этом см. раздел «Разрешение ситуации гонок» в главе 23 «Защита данных».

Переносимым интерфейсом Perl для блокировки является функция `flock` (*HANDLE, FLAGS*), описываемая в главе 29 «Функции». Perl добивается максимальной переносимости путем использования самых простых и наиболее распространенных возможностей блокировки, имеющихся на возможно большем числе платформ. Эта семантика достаточно проста для эмуляции в большинстве систем, в том числе таких, которые не поддерживают обычный системный вызов с таким именем, например, System V или Windows NT. (Если у вас система Microsoft более ранняя, чем NT, то вам, вероятно, не повезло, как и в случае, если у вас система Apple более ранняя, чем Mac OS X.)

Есть две разновидности блокировок: блокировки с совместным (shared) доступом (флаг `LOCK_SH`) и блокировки с монопольным, или взаимоисключающим (exclusive), доступом (флаг `LOCK_EX`). Несмотря на мысли, вызываемые словом «монопольный», процессы не обязаны подчиняться блокировкам файлов. Смысл в том, что `flock` реализует только *рекомендательную блокировку*, и блокировка не отнимает у других процессов возможности чтения файла или даже записи в него. Запрос монопольной блокировки – это способ, которым процесс позволяет операционной системе приостановить его выполнение, пока все текущие блокировки, будь они с совместным доступом или монопольные, не будут завершены. Аналогично, когда процесс запрашивает блокировку с совместным доступом, он просто приостанавливает свое выполнение, пока не будут сняты все монопольные блокировки. Безопасный совместный доступ к файлу может осуществляться только в случае применения механизма блокировки файлов всеми сторонами.

Поэтому `flock` по умолчанию является блокирующей операцией. Это означает, что если требуемую блокировку нельзя получить немедленно, то операционная система приостанавливает ваш процесс, пока это не будет возможно. Вот как получить блокировку с совместным доступом, применяемую обычно для чтения файла:

```
use Fcntl qw(:DEFAULT :flock);
open(FH, "< filename")  or die "невозможно открыть файл с именем: $!";
flock(FH, LOCK_SH)      or die "невозможно блокировать файл с именем: $!";
# теперь выполнять чтение из FH
```

Можно попытаться получить блокировку в неблокирующем режиме, включив флаг `LOCK_NB` в запрос `flock`. Если получить блокировку немедленно не удается, выполнение функции оказывается неудачным, и она сразу возвращает false. Вот пример:

```
flock(FH, LOCK_SH | LOCK_NB)
    or die "невозможно блокировать файл с именем: $!";
```

Можно попытаться сделать что-то другое, не создавая исключительной ситуации, как в этом примере, но, конечно, не вздумайте производить с файлом операции ввода/вывода. Если в блокировке отказано, то, не получив ее, не следует обращаться к файлу. Неизвестно, в каком искаженном состоянии вы получите файл. Основная цель неблокирующего режима в том, чтобы дать вам возможность «отойти в сторону» и сделать что-то другое в период ожидания. Но он также позволяет организовать более дружественное взаимодействие с пользователями, предупреждая их, что для получения блокировки может понадобиться время, чтобы они не чувствовали себя брошенными на произвол судьбы:

```
use Fcntl qw(:DEFAULT :flock);
open(FH, "< filename")  or die " невозможно открыть файл с именем: $!";
unless (flock(FH, LOCK_SH | LOCK_NB)) {
    local $| = 1;
    print "Ждем блокировки файла...";
    flock(FH, LOCK_SH)  or die " невозможно заблокировать файл с именем: $!";
    print "got it.\n"
}
# теперь читать FH
```

Очень может быть, что кто-нибудь захочет поместить такую неблокирующую блокировку в цикл. Основная проблема с неблокирующим режимом в том, что к тому моменту, когда вы снова выполните проверку, кто-нибудь еще может заблокировать файл, потому что вы отошли от своего места в очереди. Иногда приходится просто стоять и ждать. Если повезет, то можно почитать какие-нибудь журналы.

Блокируются указатели файлов, а не их имена.[1] При закрытии файла блокировка автоматически снимается, будь оно произведено явно с помощью вызова `close` или косвенно путем повторного открытия указателя или завершения процесса.

При получении монопольной блокировки, обычно используемой для записи, нужно проявлять еще бо́льшую осторожность. Для этого не годится обычный `open`; в случае применения режима открытия `<` происходит ошибка, если файл еще не существует, а установив режим `>`, можно разрушить существующие файлы. Вместо этого, для того чтобы заблокировать файл перед тем как его переписать, используйте `sysopen`. Успешно открыв файл для записи и еще

[1] Фактически блокируются не указатели (filehandles), а дескрипторы файлов (file descriptors), ассоциируемые с указателями, т. к. операционная система не знает указателей файлов. Это означает, что все наши сообщения `die` о невозможности получить блокировку по имени файла технически являются неточными. Но сообщения об ошибках вида «Не могу получить блокировку файла, представляемого дескриптором файла, ассоциированным с указателем файла, исходно открытым для маршрута *filename*, хотя в данный момент *filename* может представлять совершенно другой файл, нежели наш указатель» просто собьют с толку пользователя (не говоря уже о читателе).

не тронув его, нужно получить монопольную блокировку и только *тогда* усекать файл. Теперь можно перезаписать его с новыми данными.

```
use Fcntl qw(:DEFAULT :flock);
sysopen(FH, "filename", O_WRONLY | O_CREAT)
    or die "невозможно открыть файл с именем: $!";
flock(FH, LOCK_EX)
    or die "невозможно заблокировать файл с именем: $!";
truncate(FH, 0)
    or die "невозможно сделать усечение файла с именем: $!";
# now write to FH
```

Если вам нужно изменить некоторую часть файла, опять-таки используйте `sysopen`. На этот раз вы запрашиваете доступ по чтению и записи одновременно, при необходимости создавая файл. Когда файл открыт, то перед тем как осуществлять чтение или запись, нужно получить монопольную блокировку и сохранять ее на все время операции. Снять блокировку лучше всего путем закрытия файла, так как это гарантирует запись всех буферов перед снятием блокировки.

При обновлении производится чтение старых значений и запись новых. Обе операции нужно выполнять во время действия одной монопольной блокировки, чтобы другой процесс не прочел (неизбежно неверные) данные после (или даже прежде) того, как это сделаете вы, но перед тем как вы закончите запись. (Мы вернемся к этой ситуации при рассказе о совместно используемой памяти далее в этой главе.)

```
use Fcntl qw(:DEFAULT :flock);

sysopen(FH, "counterfile", O_RDWR | O_CREAT)
    or die "невозможно открыть counterfile: $!";
flock(FH, LOCK_EX)
    or die "невозможно блокировать по записи counterfile: $!";
$counter = <FH> || 0;  # в первый раз будет undef
seek(FH, 0, 0)
    or die "невозможно вернуться к началу counterfile : $!";
print FH $counter+1, "\n"
    or die "невозможно осуществить запись в counterfile: $!";

# следующая строка в данной программе технически избыточна, но
# в общем случае мысль правильная
truncate(FH, tell(FH))
    or die " невозможно выполнить усечение counterfile: $!";
close(FH)
    or die " невозможно закрыть counterfile: $!";
```

Нельзя заблокировать файл, который вы еще не открыли, и нельзя применить одну блокировку к нескольким файлам. Что, однако, можно сделать, – так это использовать совершенно отдельный файл в качестве «семафора», чтобы обеспечить управление доступом к чему-то еще посредством обычных блокировок этого файла «семафора» с совместным и монопольным доступом. У такого подхода есть несколько преимуществ. Можно иметь один файл блокировки, управляющий доступом к нескольким файлам, избегая

при этом такого взаимного блокирования (deadlock), которое происходит, когда один процесс пытается заблокировать файлы в одном порядке, а другой – те же файлы в другом порядке. Можно использовать файл «семафора» для блокировки целого каталога с файлами. Можно даже управлять доступом к тому, что не находится в файловой системе, например, объекту в памяти с совместным доступом или сокету, по которому несколько полученных путем ветвления серверов намерены выполнить вызов accept.

Если у вас есть файл DBM, не предоставляющий собственного механизма блокировки, то одновременным доступом к нему со стороны нескольких агентов лучше всего управлять с использованием вспомогательного файла блокировки. В противном случае внутреннее кэширование вашей библиотеки DBM может рассинхронизироваться с дисковым файлом. Прежде чем вызывать dbmopen или tie, откройте и заблокируйте файл «семафора». Если вы открываете базу данных с ключом O_RDONLY, то для блокировки потребуется установка флага LOCK_SH. В противном случае используйте LOCK_EX для монопольного доступа для обновления базы данных. (Это действенно, только если все участники действия согласны обращать внимание на «семафор».)

```
use Fcntl qw(:DEFAULT :flock);
use DB_File;  # только в целях демонстрации; годится любая БД

$DBNAME  = "/path/to/database";
$LCK     = $DBNAME . ".lockfile";

# используйте O_RDWR, если собираетесь что-то поместить в файл блокировок
sysopen(DBLOCK, $LCK, O_RDONLY | O_CREAT)
    or die "невозможно открыть $LCK: $!";

# нужно получить блокировку перед открытием базы данных
flock(DBLOCK, LOCK_SH)
    or die "невозможно LOCK_SH $LCK: $!";

tie(%hash, "DB_File", $DBNAME, O_RDWR | O_CREAT)
    or die "невозможно tie $DBNAME: $!";
```

Теперь можно без опасения совершать любые действия со связанным с базой данных %hash. Закончив работу с базой данных, обеспечьте явное закрытие этих ресурсов, причем в порядке, обратном тому, в котором вы их получили:

```
untie %hash;    # закрыть базу данных прежде файла блокировки
close DBLOCK;   # теперь безопасно снять блокировку
```

Если у вас установлена библиотека GNU DBM, можно использовать неявную блокировку из стандартного модуля GDBM_File. Если только начальный tie не содержит флага GDBM_NOLOCK, библиотека разрешает одновременно открывать файл GDBM для записи только одному процессу и запрещает одновременно открывать базу данных записывающим и читающим процессам.

Передача указателей файлов

При создании дочернего процесса с помощью fork новый процесс наследует все открытые указатели файлов родителя. Использование указателей фай-

лов для межпроцессного взаимодействия легче всего сначала проиллюстрировать на обычных файлах. Важно разобраться в том, как это работает, чтобы овладеть более сложными механизмами каналов и сокетов, описываемыми далее в этой главе.

В этом простейшем примере открывается файл и запускается дочерний процесс. Затем дочерний запрос использует указатель файла, уже открытый для него:

```
open(INPUT, "< /etc/motd")      or die "/etc/motd: $!";
if ($pid = fork) { waitpid($pid,0) }
else {
    defined($pid)         or die "fork: $!";
    while (<INPUT>) { print "$.: $_" }
    exit;  # не позволить дочернему процессу вернуться в основной код
}
# Теперь указатель INPUT в родителе находится в EOF
```

Доступ к файлу, предоставленный посредством open, сохраняется до закрытия указателя файла; изменения прав доступа к файлу и прав доступа владельца не оказывают влияния на доступ. Даже если процесс позже изменит свои ID пользователя или группы либо изменятся пользователь или группа, являющиеся владельцем файла, это не окажет влияния на уже открытые указатели файлов. Программы, выполняющиеся с большими правами (такие, как системные демоны или программы, устанавливающие id), часто открывают файл, а затем передают указатель дочернему процессу, который не смог бы открыть файл самостоятельно.

Хотя эта возможность очень удобна в случае ее целевого применения, она может также создавать проблемы с защитой данных, когда указатели файлов случайно проникают из одной программы в другую. Чтобы избежать неявной передачи доступа к любым указателям файлов, Perl автоматически закрывает любые открытые им указатели файлов (в том числе каналы и сокеты) при явном запуске новой программы через exec или неявном выполнении ее через вызов конвейеризованного open, system или qx// (обратные штрихи). Системные указатели файлов STDIN, STDOUT и STDERR являются исключением, поскольку их основная цель – обеспечить связь между программами. Поэтому одним из способов передачи указателя файла новой программе является его копирование в один из стандартных файловых указателей (filehandles):

```
open(INPUT, "< /etc/motd")      or die "/etc/motd: $!";
if ($pid = fork) { wait }
else {
    defined($pid)               or die "fork: $!";
    open(STDIN, "<&INPUT")      or die "dup: $!";
    exec("cat", "-n")           or die "exec cat: $!";
}
```

Если в действительности требуется, чтобы новая программа получила доступ к указателю файла, отличному от этих трех, это можно осуществить, но

при этом нужно проделать одну из двух вещей. Когда Perl открывает новый файл (или канал, или сокет), он проверяет текущее значение переменной $^F ($SYSTEM_FD_MAX). Если числовой дескриптор файла, используемый этим новым указателем файла, больше $^F, то дескриптор помечается как подлежащий закрытию. В противном случае Perl оставляет его в покое, и выполняемые вами новые программы унаследуют доступ.

Не всегда легко предсказать, какой дескриптор будет иметь вновь открываемый указатель файла, но можно временно установить максимальный дескриптор файла в системе равным какому-нибудь очень большому числу на время открытия:

```
# открыть файл и пометить INPUT как оставленный открытым на время выполнения exec
{
    local $^F = 10_000;
    open(INPUT, "< /etc/motd")   or die "/etc/motd: $!";
} # прежнее значение $^F восстанавливается при выходе из области видимости
```

Теперь необходимо заставить новую программу обратить внимание на номер дескриптора только что открытого указателя файла. Самое правильное решение (для систем, которые это поддерживают) состоит в передаче специального имени файла, соответствующего дескриптору файла. Если в системе есть каталог с именем */dev/fd* или */proc/$$/fd*, содержащий файлы с номерами от 0 до максимального числа поддерживаемых дескрипторов, вам, возможно, удастся воспользоваться такой стратегией. (Во многих системах Linux есть оба каталога, но только */proc* содержит то, что нужно. Так, в BSD и Solaris это каталог */dev/fd*. Вам придется поискать в системе, где они расположены.) Сначала откройте и пометьте указатель файла как открытый для exec аналогично приведенному выше коду, затем выполните ветвление подобно следующему:

```
if ($pid = fork) { wait }
else {
    defined($pid)                or die "fork: $!";
    $fdfile = "/dev/fd/" . fileno(INPUT);
    exec("cat", "-n", $fdfile)   or die "exec cat: $!";
}
```

Если в вашей системе поддерживается системный вызов fcntl, можете вручную изменить флаг закрытия указателя файла при выполнении exec. Это удобно в тех случаях, когда при создании указателя файла было неизвестно, что его потребуется использовать совместно с дочерними процессами.

```
use Fcntl qw/F_SETFD/;

fcntl(INPUT, F_SETFD, 0)
    or die "Не могу сбросить флаг для закрытия при exec для INPUT: $!\n";
```

Можно также заставить указатель файла закрываться:

```
fcntl(INPUT, F_SETFD, 1)
    or die " Не могу установить флаг для закрытия при exec для INPUT: $!\n";
```

А можно и запросить текущее состояние:

```
use Fcntl qw/F_SETFD F_GETFD/;

printf("INPUT будет %s для exec\n",
    fcntl(INPUT, F_GETFD, 1) ? "закрыт" : "открыт слева");
```

Если система не поддерживает дескрипторы файлов, указанные в файловой системе, а нужно передать указатель файла, отличный от STDIN, STDOUT или STDERR, это можно обойти, но для этого в программе придется предусмотреть дополнительные действия. Обычно данная задача решается передачей номера дескриптора через переменную окружения или в качестве параметра командной строки.

Если исполняемая программа написана на Perl, то преобразовать дескриптор файла в указатель можно посредством open. Вместо задания имени файла используйте &=, а затем номер дескриптора.

```
if (defined($ENV{input_fdno}) && $ENV{input_fdno}) =~ /^\d$/) {
    open(INPUT, "<&=$ENV{input_fdno}")
        or die "невозможно fdopen $ENV{input_fdno} для ввода: $!";
}
```

Еще проще, если вы собираетесь запускать подпрограмму или программу Perl, которая ждет имя файла в качестве аргумента. Можно прибегнуть к открытию файла по дескриптору обычной функции Perl open (но не sysopen или open с тремя аргументами), чтобы это происходило автоматически. Представьте себе, что у вас есть такая несложная программа на Perl:

```
#!/usr/bin/perl -p
# nl - число строк ввода
printf "%6d  ", $.;
```

Предполагая, что вы настроили указатель INPUT так, чтобы он оставался открытым при выполнении exec, можно вызывать эту программу следующим образом:

```
$fdspec = '<&=' . fileno(INPUT);
system("nl", $fdspec);
```

или перехватывать вывод:

```
@lines = `nl '$fdspec'`;  # одинарные кавычки защищают spec от оболочки
```

Запускаете вы другую программу или нет, при использовании дескрипторов файлов, наследуемых через fork, существует одна маленькая неувязочка (gotcha). В отличие от переменных, которые копируются при выполнении fork, становясь фактически идентичными, но независимыми копиями, указатели файлов в обоих процессах *действительно* одни и те же. Если какой-то процесс читает данные из указателя, то позиция в файле в другом процессе также перемещается вперед, и эти данные становятся недоступными в обоих процессах. Если процессы выполняют чтение поочередно, то в этом файле

будет иметь место своеобразная чехарда. Это может быть интуитивно понятно в случае указателей, работающих с устройствами последовательного доступа, каналами и сокетами, поскольку это устройства только для чтения с временными данными, но такая работа с дисковыми файлами может оказаться неожиданной. Если в результате возникают проблемы, после выполнения `fork` откройте заново файлы, которые требуют раздельного просмотра.

Оператор `fork` является наследием, полученным от Unix, и это значит, что он может некорректно реализовываться на платформах, отличных от Unix/POSIX. Примечательно, что `fork` работает на системах Microsoft только для Perl 5.6 (или выше) под Windows 98 (или выше). Хотя в этих системах `fork` реализуется посредством множественных одновременных потоков исполнения внутри одной программы, это не те потоки, в которых по умолчанию совместно используются все данные; в данном случае это относится только к указателям файлов. См. также главу 17 «Потоки».

Каналы

Канал (*pipe*) является однонаправленным каналом ввода-вывода, способным передавать поток байтов от одного процесса к другому. Каналы могут быть как именованными, так и безымянными. Вам, вероятно, лучше знакомы безымянные каналы, поэтому с них мы и начнем.

Анонимные каналы

Функция Perl `open` открывает канал, а не файл, если перед вторым аргументом стоит символ конвейера. В результате оставшиеся аргументы превращаются в команду, которая будет интерпретирована как процесс (или группа процессов), в который нужно передать или из которого получить поток данных. Ниже приведен способ запуска дочернего процесса, в который предполагается осуществлять запись:

```
open SPOOLER, "| cat -v | lpr -h 2>/dev/null"
    or die "can't fork: $!";
local $SIG{PIPE} = sub { die "spooler pipe broke" };
print SPOOLER "stuff\n";
close SPOOLER or die "bad spool: $! $?";
```

В этом примере фактически запускаются два процесса, в первый из которых (*cat*) вывод осуществляется напрямую. Затем второй процесс (*lpr*) получает выходной поток первого процесса. В программировании оболочек это часто называется *конвейером* (*pipeline*). В конвейере может находиться подряд сколько угодно процессов, лишь бы те, которые находятся в середине, могли работать в режиме *фильтра*, то есть читать из стандартного ввода и выводить в стандартный вывод.

Perl использует системную оболочку по умолчанию (*/bin/sh* в Unix), если команда конвейера содержит специальные символы, распознаваемые оболоч-

кой. Если запускается только одна команда и нет необходимости в обращении к оболочке, то можно использовать формат открытия канала с несколькими аргументами:

```
open SPOOLER, "|-", "lpr", "-h"    # требует 5.6.1
    or die "can't run lpr: $!";
```

Если повторно открыть стандартное устройство вывода программы как канал в другую программу, то все, что будет выведено после этого в STDOUT, поступит на стандартный ввод новой программы. Поэтому для организации постраничного вывода программы[1] следует использовать:

```
if (-t STDOUT) {             # только если stdout является терминалом
    my $pager = $ENV{PAGER} || 'more';
    open(STDOUT, "| $pager")    or die "невозможно запустить пейджер: $!";
}
END {
    close(STDOUT)               or die "невозможно закрыть STDOUT: $!"
}
```

Если вы записываете в указатель файла, соединенный с каналом, всегда явно закрывайте указатель, закончив с ним работу. Благодаря этому ваша основная программа не завершит работу раньше своих потомков.

Вот как запустить дочерний процесс, из которого предполагается осуществлять чтение:

```
open STATUS, "netstat -an 2>/dev/null |"
    or die "can't fork: $!";
while (<STATUS>) {
    next if /^(tcp|udp)/;
    print;
}
close STATUS or die "bad netstat: $! $?";
```

Открыть многоступенчатый канал для ввода можно так же, как для вывода. И, как и раньше, можно обойти оболочку, используя альтернативную форму open:

```
open STATUS, "-|", "netstat", "-an"      # требует 5.6.1
    or die "can't run netstat: $!";
```

Но при этом вы не получите переадресации ввода/вывода, разыменования групповых символов и многоступенчатых конвейеров, поскольку Perl полагается при их осуществлении на оболочку.

Возможно, вы заметили, что можно использовать обратный штрих для достижения того же эффекта, что и открытие канала для чтения:

```
print grep { !/^(tcp|udp)/ } `netstat -an 2>&1`;
die "bad netstat" if $?;
```

[1] То есть, чтобы просматривать по одному экрану за раз.

Хотя обратный штрих чрезвычайно удобен, при данной операции приходится считывать в память сразу все данные, поэтому часто открытие своего указателя файла для канала и обработка файла по одной строчке или записи оказываются более эффективными. Это обеспечивает более тонкий контроль над всей операцией, позволяя досрочно убить дочерний процесс, если вы этого захотите. Большей эффективности можно добиться и благодаря обработке вводимых данных во время их поступления, т. к. компьютеры могут чередовать разные операции при одновременном выполнении двух или более процессов. (Даже на машине с одним центральным процессором операции ввода и вывода могут происходить, когда процессор занят чем-то другим.)

Из-за того что одновременно выполняются два или более процессов, катастрофа может постичь дочерний процесс в любой момент между `open` и `close`. Это означает, что родитель должен проверять значения, возвращаемые как `open`, так и `close`. Проверки только `open` недостаточно, поскольку она может лишь сообщить об успехе `fork` и, может быть, что следующая команда была успешно запущена. (Это можно узнать только в новых версиях Perl, и только если команда выполнена непосредственно ответившимся потомком, а не через оболочку.) Обо всех катастрофах, происходящих после этого, потомок сообщает родителю в виде ненулевого кода возврата. Когда его видит функция `close`, она возвращает значение «ложь», указывающее на то, что истинное значение состояния нужно прочесть из переменной `$?` (`$CHILD_ERROR`). Поэтому проверка значения, возвращаемого `close`, столь же важна, как проверка `open`. Осуществляя запись в канал, следует приготовиться к обработке сигнала `PIPE`, который посылается, если процесс на другом конце умрет раньше, чем будет закончена отправка ему данных.

Разговор с самим собой

Другой подход к IPC состоит в том, чтобы заставить программу, так сказать, разговаривать с самой собой. В действительности ваш процесс общается через каналы с собственной ответившейся копией. Этот механизм действует во многом как открытие канала, о котором мы говорили в предыдущем разделе, за исключением того, что дочерний процесс продолжает выполнение вашего сценария, а не какой-либо другой команды.

Чтобы представить это функции `open`, используется псевдокоманда, состоящая из знака «минус». Поэтому второй аргумент `open` выглядит как `"-|"` или `"|-"` в зависимости от того, хотите вы направить конвейер от себя или к себе. Как и обычная команда `fork`, функция `open` возвращает ID дочернего процесса в родительский процесс и `0` – в дочерний процесс. Другая асимметрия состоит в том, что указатель файла, объявленный в `open`, используется только в родительском процессе. Дочерний конец канала прикрепляется к `STDIN` или `STDOUT`. То есть, если вы открываете канал `TO` (к минусу) с помощью `"|-"`, то можете выводить в открытый указатель файла то, что ваш потомок найдет в `STDIN`:

```
if (open(TO, "|-")) {
    print TO $fromparent;
```

```
}
else {
    $tochild = <STDIN>;
    exit;
}
```

Если открыть канал FROM (от минуса) с помощью "-|", можно читать из открытого дескриптора файла (filehandle), который возвратит то, что ваш потомок запишет в STDOUT:

```
if (open(FROM, "-|")) {
    $toparent = <FROM>;
}
else {
    print STDOUT $fromchild;
    exit;
}
```

Распространенным применением этой конструкции является обход оболочки, когда необходимо открыть канал из команды. Это может потребоваться, если нежелательно, чтобы оболочка интерпретировала метасимволы, встречающиеся в именах файлов, передаваемых команде. Версия Perl 5.6.1 (или более высокая) позволяет для получения того же результата использовать формат open с несколькими аргументами.

Другое применение open, выполняющего ветвление, состоит в безопасном открытии файла или команды, даже если вы работаете под необходимым UID или GID. Создаваемый fork потомок отбрасывает любые дополнительные права доступа, затем безопасно открывает файл или команду и действует как посредник, передавая данные между своим более мощным родителем и файлом или командой, которые он открыл. Примеры можно найти в разделе «Доступ к командам и файлам, когда права сокращены» главы 23 «Защита данных».

Одно из творческих применений открытия с разделением на процессы состоит в фильтрации собственного вывода. Некоторые алгоритмы значительно проще реализовать за два отдельных прохода, чем за один. Вот простой пример, в котором программа Unix *tee*(1) эмулируется путем перенаправления обычного вывода в конвейер. Агент, находящийся на другом конце конвейера (одна из наших собственных подпрограмм), распределяет вывод во все указанные файлы:

```
tee("/tmp/foo", "/tmp/bar", "/tmp/glarch");

while (<>) {
    print "$ARGV at line $. => $_";
}
close(STDOUT)  or die "невозможно закрыть STDOUT: $!";

sub tee {
    my @output = @_;
    my @handles = ();
    for my $path (@output) {
```

```
        my $fh;
        unless (open ($fh, ">", $path)) {
            warn "невозможно записать в $path: $!";
            next;
        }
        push @handles, $fh;
    }

    # повторно открыть STDOUT в родителе и вернуться
    return if my $pid = open(STDOUT, "|-");
    die "невозможно выполнить ветвление: $!" unless defined $pid;

    # процесс STDIN в дочернем процессе
    while (<STDIN>) {
        for my $fh (@handles) {
            print $fh $_ or die "отказ при выводе из tee: $!";
        }
    }
    for my $fh (@handles) {
        close($fh) or die "отказ при закрытии tee: $!";
    }
    exit;  # не позволяйте потомку возвращаться в основную программу!
}
```

Этот прием можно применять повторно, чтобы поместить в выходной поток столько фильтров, сколько нужно. Просто продолжайте вызывать функции, открывающие STDOUT с ветвлением процессов, и пусть дочерний процесс читает из родительского (который он видит как STDIN) и передает обработанный вывод далее к следующей функции потока.

Другим интересным применением «разговора с самим собой» с помощью открытия с делением на процессы является перехват вывода «плохо воспитанной» функции, которая постоянно выводит свои результаты в STDOUT. Представьте себе, что в Perl есть только printf, а sprintf нет. Вам тогда нужно что-то, работающее как обратный штрих, но с функциями Perl, а не с внешними командами:

```
badfunc("arg");                        # черт, убежала!
$string = forksub(\&badfunc, "arg");   # перехватить как строку
@lines  = forksub(\&badfunc, "arg");   # как отдельные строки

sub forksub {
    my $kidpid = open my $self, "-|";
    defined $kidpid         or die "невозможно ветвление: $!";
    shift->(@_), exit       unless $kidpid;
    local $/                unless wantarray;
    return <$self>;         # закрывается при выходе из области видимости
}
```

Этот код не претендует на эффективность; связанный указатель файла работал бы, вероятно, значительно быстрее. Но написать его намного легче, когда вы находитесь в большей спешке, чем ваш компьютер.

Двунаправленная связь

Использование функции open для объединения с другой командой через канал достаточно хорошо работает при однонаправленной связи, но как организовать двунаправленный обмен? Подход, который кажется очевидным, на самом деле не работает:

```
open(PROG_TO_READ_AND_WRITE, "| некоторая программа |")  # НЕВЕРНО!
```

а если вы забыли включить вывод предупредительных сообщений, то полностью пропустите следующее диагностическое сообщение:

```
Can't do bidirectional pipe at myprog line 3.
[Невозможно создать двунаправленный канал в myprog строка 3]
```

Функция open не позволяет сделать это, потому что подвержена взаимоблокировкам, если не проявлять большую осторожность. Но если очень хочется, то можно использовать стандартный библиотечный модуль IPC::Open2, чтобы подключить два канала к STDIN и STDOUT подпроцесса. Имеется также модуль IPC::Open3 для тринаправленного ввода/вывода (позволяющий перехватывать и STDERR дочернего процесса), но для него требуется либо «неуклюжий» цикл select, либо более удобный модуль IO::Select. Однако тогда придется избегать буферизованных операций ввода в Perl, таких как <> (readline).

Вот пример использования open2:

```
use IPC::Open2;
local (*Reader, *Writer);
$pid = open2(\*Reader, \*Writer, "bc -l");
$sum = 2;
for (1 .. 5) {
    print Writer "$sum * $sum\n";
    chomp($sum = <Reader>);
}
close Writer;
close Reader;
waitpid($pid, 0);
print "сумма равна $sum\n";
```

Можно также вызывать самооживление лексических дескрипторов файлов:

```
my ($fhread, $fhwrite);
$pid = open2($fhread, $fhwrite, "cat -u -n");
```

Проблема здесь, в целом, связана с тем, что стандартная буферизация ввода/вывода испортит вам всю жизнь. Хотя для дескриптора файла вывода производится автоматическая очистка буфера (это делает вместо вас библиотека), и поэтому процесс на другом конце получает данные своевременно, обычно никакими силами нельзя заставить его совершить взаимную любезность. В данном конкретном случае нам повезло: bc рассчитывает на работу через канал (pipe) и умеет очищать буфер для каждой выводимой строки. Но так работает не много команд, поэтому такой способ редко срабатывает, если только не

вы сами написали программу, находящуюся на другом конце двунаправленного канала. Даже простые и, очевидно, интерактивные программы типа *ftp* здесь не срабатывают, потому что не буферизируют строку для канала; они делают это только с устройством tty. Помощь могут оказать модули `IO::Pty` и `Expect` из CPAN, которые предоставляют устройство tty (фактически реальное псевдо-tty, но оно действует как настоящее). Благодаря этому мы получаем буферизацию в другом процессе, не модифицируя эту программу.

Если наша программа разбита на несколько процессов, и все они должны иметь возможность двустороннего общения, то из попытки использовать для этого интерфейсы каналов Perl высокого уровня ничего не выйдет, потому что все они – однонаправленные. Нам потребуются два вызова функции низкого уровня `pipe`, по одному для каждого направления передачи:

```
pipe(FROM_PARENT, TO_CHILD)     or die "pipe: $!";
pipe(FROM_CHILD,  TO_PARENT)    or die "pipe: $!";
select((select(TO_CHILD), $| = 1))[0]);   # автоочистка
select((select(TO_PARENT), $| = 1))[0]);  # автоочистка

if ($pid = fork) {
    close FROM_PARENT; close TO_PARENT;
    print TO_CHILD "Родительский процесс Pid $$ отправляет это\n";
    chomp($line = <FROM_CHILD>);
    print "Родительский процесс Pid $$ только что прочел: `$line'\n";
    close FROM_CHILD; close TO_CHILD;
    waitpid($pid,0);
} else {
    die "невозможно ветвление: $!" unless defined $pid;
    close FROM_CHILD; close TO_CHILD;
    chomp($line = <FROM_PARENT>);
    print "Дочерний процесс Pid $$ только что прочел это: `$line'\n";
    print TO_PARENT "Дочерний процесс Pid $$ отправляет это\n";
    close FROM_PARENT; close TO_PARENT;
    exit;
}
```

Во многих системах Unix на практике не требуется организовывать два отдельных обращения к `pipe`, чтобы установить полнодуплексную связь между родительским и дочерним процессами. Системный вызов `socketpair` обеспечивает двусторонние соединения между связанными процессами на одной и той же машине. Поэтому вместо двух каналов нужна одна пара сокетов.

```
use Socket;
socketpair(Child, Parent, AF_UNIX, SOCK_STREAM, PF_UNSPEC)
    or die "socketpair: $!";

# или позвольте Perl выбрать имена файлов за вас
my ($kidfh, $dadfh);
socketpair($kidfh, $dadfh, AF_UNIX, SOCK_STREAM, PF_UNSPEC)
    or die "socketpair: $!";
```

После порождения дочернего процесса родительский процесс закрывает указатель `Parent` и осуществляет чтение и запись через указатель `Child`. Меж-

ду тем, дочерний процесс закрывает указатель `Child`, после чего читает и записывает через указатель `Parent`.

Если двунаправленная связь интересует вас, потому что процесс, с которым вы хотите общаться реализован как стандартный сервис Интернета, то, вероятно, следует отказаться от посредников и использовать модуль CPAN, предназначенный специально для этой цели. (Перечень некоторых из них см. в разделе «Сокеты».)

Именованные каналы

Именованный канал (named pipe), часто называемый FIFO, является механизмом для организации обмена данными между неродственными процессами на одной и той же машине. Имена в «именованном» канале существуют в файловой системе, и это просто необычный способ сказать, что в пространство имен файловой системы можно поместить особый файл, за которым стоит не диск, а некий процесс.[1]

FIFO удобно использовать, когда необходимо соединиться с неродственным процессом. При открытии FIFO ваш процесс блокируется, пока не появится процесс на другом конце. Поэтому, если читающий процесс открывает FIFO первым, он блокируется, пока не покажется пишущий процесс, – и наоборот.

Для создания именованного канала используйте функцию POSIX `mkfifo` – разумеется, если ваша система поддерживает POSIX. Пользователям систем Microsoft понадобится заглянуть в модуль `Win32::Pipe`, который, несмотря на то, что кажется обратное, создает именованные каналы. (Пользователи Win32 создают анонимные каналы с помощью `pipe`, как и все мы.)

Допустим, например, что каждое новое чтение нашего файла *.signature* должно приводить к разным результатам. Сделайте его именованным каналом, на другом конце которого сидит программа Perl и «выплевывает» случайным образом выбранные мудрые изречения. Теперь, когда любая программа (почтовая, для чтения телеконференций, finger и т. д.) попытается прочесть этот файл, она подключится к нашей программе и прочтет динамически создаваемую подпись.

В следующем примере мы используем редко встречающийся оператор проверки файла `"-p"`, чтобы проверить, не удалена ли случайно кем-нибудь (или чем-нибудь) наша FIFO.[2] Если да, то нечего и пытаться открыть ее, и мы рассматриваем это как условие завершения. Если бы мы вызвали простую функцию `open` в режиме `"> $fpath"`, то возникла бы ситуация, чреватая риском перезаписи *.signature* как обычного файла, если бы он пропал между проверкой `"-p"` и открытием. Режим `"+< $fpath"` тоже не годится, потому что открытие FIFO для чтения-записи не является блокирующим (это верно только

---

[1] То же можно делать с сокетами *домена* Unix, но с ними нельзя использовать `open`.

[2] Другое применение состоит в проверке того, что некоторый указатель соединен с каналом, именованным или безымянным, например `-p STDIN`.

для FIFO). Использовав sysopen и опустив флаг O_CREAT, мы обойдем эту проблему и никогда случайно не создадим файл.

```
use Fcntl;              # для sysopen
chdir;                  # в исходный каталог
$fpath = `.signature`;
$ENV{PATH} .= ":/usr/games";

unless (-p $fpath) {    # не канал,
    if (-e _) {         # а что-то иное
        die "$0: не буду перезаписывать .signature\n";
    } else {
        require POSIX;
        POSIX::mkfifo($fpath, 0666) or die "can't mknod $fpath: $!";
        warn "$0: создан $fpath как именованный канал\n";
    }
}

while (1) {
    # выйти, если файл signature удален вручную
    die "Файл канала исчез" unless -p $fpath;
    # следующая строка блокирует, пока не появится читатель
    sysopen(FIFO, $fpath, O_WRONLY)
        or die "невозможность записи $fpath: $!";
    print FIFO "John Smith (smith\@host.org)\n", `fortune -s`;
    close FIFO;
    select(undef, undef, undef, 0.2);  # спать 1/5 секунды
}
```

Небольшая пауза после закрытия нужна, чтобы дать читающему процессу возможность прочесть то, что записано. Если сразу повторить цикл и снова открыть FIFO, прежде чем читающий процесс закончит чтение только что переданных данных, то он не увидит конца файла, так как снова работает пишущий процесс. Так мы и будем ходить по кругу, пока на какой-нибудь итерации пишущий процесс немного не отстанет и читающий процесс, наконец, не увидит этот неуловимый конец файла. (А мы беспокоились о ситуации гонки!)

System V IPC

System V IPC ненавидят все. Она работает медленнее, чем бумажная лента, вырезает коварные маленькие пространства имен, совершенно не связанные с файловой системой, использует для именования своих объектов враждебные восприятию числа и постоянно теряет ход собственных мыслей. Очень часто перед системным администратором встает задача «найти и уничтожить», чтобы выследить потерянные объекты SysV IPC с помощью *ipcs*(1) и убить их с помощью *ipcrm*(1) раньше, чем кончится системная память.

Несмотря на все эти муки, IPC древней SysV все же имеет ряд возможных применений. Перечислим три типа объектов IPC: совместно используемая

память, семафоры и сообщения. Для передачи сообщений предпочтительным механизмом сегодня являются сокеты, которые также значительно лучше переносимы. Для простых семафоров обычно используется файловая система. А что касается совместно используемой памяти, то это решать вам. Если она у вас есть, то подойдет современный вызов *mmap*(2),[1] хотя качество его реализации зависит от используемой системы. Требуется также проявлять некоторую осторожность, чтобы Perl не переместил ваши строки из того места, куда их положил *mmap*(2). Но когда программисты подумывают о применении *mmap*(2), то слышат бессвязное бормотание местных знатоков о том, что он сильно страдает от деликатных проблем связности кэша в системах, где отсутствует нечто, что они называют «кэш с унифицированным буфером» (unified buffer cache), или, возможно, «общая шина-мухоловка» (flycatcher unibus), и тогда, решив, что знакомый черт лучше незнакомого, быстренько возвращаются к знакомым и ненавидимым средствам SysV IPC для организации работы с совместно используемой памятью.

Вот небольшая программа, демонстрирующая управляемый доступ к буферу совместно используемой памяти целого «выводка» процессов-братьев. Объекты SysV IPC могут быть совместно доступны и *неродственным* процессам на одном и том же компьютере, но тогда надо решать, как они будут искать друг друга. Чтобы содействовать безопасному доступу, мы создадим для каждого по семафору.[2]

Каждый раз, когда нужно получить из совместно используемой памяти новое значение или поместить его туда, необходимо сначала пройти семафор. Это может стать довольно утомительным, поэтому создадим для доступа оболочку в виде объекта. Модуль `IPC::Shareable` делает еще один шаг вперед, оборачивая свой класс объектов в интерфейс `tie`.

Эта программа выполняется, пока вы не прервете ее с помощью комбинации <Ctrl>+<C> или эквивалентной:

```
#!/usr/bin/perl -w
use v5.6.0;   # или  старшую
use strict;
use sigtrap qw(die INT TERM HUP QUIT);
my $PROGENY = shift(@ARGV) || 3;
eval { main() };   # смотрите ниже в DESTROY, зачем это нужно
die if $@ && $@ !~ /^Caught a SIG/;
print "\nDone.\n";
exit;
```

---

[1] Есть даже модуль `Mmap` в CPAN.

[2] Более реалистичным было бы создание пары семафоров для каждой области памяти совместного доступа: один для чтения и другой для записи. Фактически это и делает модуль `IPC::Shareable` из CPAN. Но мы стараемся сохранить простоту. Стоит, однако, признать, что с помощью пары семафоров можно было бы воспользоваться единственной, пожалуй, подкупающей особенностью SysV IPC: можно было бы осуществлять атомарные операции над целыми группами семафоров, как над одним объектом, что иногда полезно.

```
sub main {
    my $mem = ShMem->alloc("Original Creation at " . localtime);
    my(@kids, $child);
    $SIG{CHLD} = 'IGNORE';
    for (my $unborn = $PROGENY; $unborn > 0; $unborn- -) {
        if ($child = fork) {
            print "$$ begat $child\n";
            next;
        }
        die "невозможно ветвление: $!" unless defined $child;
        eval {
            while (1) {
                $mem->lock();
                $mem->poke("$$ " . localtime)
                    unless $mem->peek =~ /^$$\b/o;
                $mem->unlock();
            }
        };
        exit;  # смерть дочернего процесса
    }
    while (1) {
        print "Buffer is ", $mem->get, "\n";
        sleep 1;
    }
}
```

А вот листинг пакета ShMem, используемый в программе. Можете поместить его в конец программы или в собственный файл (с "1; " в конце) и загрузить через require из основной программы. (Два модуля IPC, используемые в нем поочередно, находятся в стандартном дистрибутиве Perl.)

```
package ShMem;
use IPC::SysV qw(IPC_PRIVATE IPC_RMID IPC_CREAT S_IRWXU);
use IPC::Semaphore;
sub MAXBUF() { 2000 }

sub alloc {    # метод конструктора
    my $class = shift;
    my $value = @_ ? shift : '';

    my $key = shmget(IPC_PRIVATE, MAXBUF, S_IRWXU) or die "shmget: $!";
    my $sem = IPC::Semaphore->new(IPC_PRIVATE, 1, S_IRWXU | IPC_CREAT)
                        or die "IPC::Semaphore->new: $!";
    $sem->setval(0,1)    or die "sem setval: $!";

    my $self = bless {
        OWNER   => $$,
        SHMKEY  => $key,
        SEMA    => $sem,
    } => $class;

    $self->put($value);
    return $self;
}
```

Теперь о методах выборки и записи. Методы get и put блокируют буфер, а peek и poke – нет, поэтому последние два должны применяться только тогда, когда объект заблокирован вручную, что приходится делать, если необходимо извлечь старое значение и записать обратно модифицированное – все в течение одной и той же блокировки. Демонстрационная программа осуществляет это в своем цикле while (1). Вся транзакция должна произойти при одной и той же блокировке, иначе проверка и установка не будут атомарными и могут стать бомбой.

```
sub get {
    my $self = shift;
    $self->lock;
    my $value = $self->peek(@_);
    $self->unlock;
    return $value;
}
sub peek {
    my $self = shift;
    shmread($self->{SHMKEY}, my $buff='', 0, MAXBUF) or die "shmread: $!";
    substr($buff, index($buff, "\0")) = '';
    return $buff;
}
sub put {
    my $self = shift;
    $self->lock;
    $self->poke(@_);
    $self->unlock;
}
sub poke {
    my($self,$msg) = @_;
    shmwrite($self->{SHMKEY}, $msg, 0, MAXBUF) or die "shmwrite: $!";
}
sub lock {
    my $self = shift;
    $self->{SEMA}->op(0,-1,0) or die "semop: $!";
}
sub unlock {
    my $self = shift;
    $self->{SEMA}->op(0,1,0) or die "semop: $!";
}
```

Наконец, классу нужен деструктор, чтобы после того как объект закончит свое существование, можно было вручную освободить совместно используемую память и семафор, хранящиеся внутри объекта. В противном случае они переживут своего создателя, и, чтобы избавиться от них, придется прибегнуть к *ipcs* и *ipcrm* (или помощи системного администратора). Вот почему мы пошли на создание сложных оболочек в основной программе, преобразующих сигналы в исключительные ситуации: чтобы выполнились все деструкторы, из памяти были удалены объекты SysV IPC, а системные администраторы не занимались нашими делами.

```
sub DESTROY {
    my $self = shift;
    return unless $self->{OWNER} == $$;  # предотвратить повторное освобождение
памяти
    shmctl($self->{SHMKEY}, IPC_RMID, 0)    or warn "shmctl RMID: $!";
    $self->{SEMA}->remove()                 or warn "sema->remove: $!";
}
```

Сокеты

Механизмы IPC, обсуждавшиеся выше, имеют одно строгое ограничение: они предназначены для связи между процессами на одном и том же компьютере. (Несмотря на то что иногда доступ к файлам может осуществляться с нескольких машин, если используются механизмы типа NFS (сетевой файловой системы), во многих реализациях NFS блокировка не работает, что лишает одновременный доступ почти всякого интереса.) Для организации сетевого взаимодействия общего назначения нужно выбирать сокеты. Хотя сокеты (sockets) были придуманы для BSD, они быстро распространились на другие виды Unix, а в наше время интерфейс сокетов можно найти почти в любой имеющейся «жизнеспособной» системе. Если на машине не поддерживаются сокеты, возникнут колоссальные трудности при работе с Интернетом.

С помощью сокетов можно использовать и виртуальные каналы (как потоки TCP), и датаграммы (как пакеты UDP). В зависимости от типа системы могут иметься и дополнительные возможности. Но самый распространенный способ программирования сокетов – использование TCP через сокеты доменов Интернета, о чем мы здесь и расскажем. Такие сокеты предоставляют надежные соединения, работающие несколько сходным с двунаправленными каналами способом, но не ограниченные локальной машиной. Два самых ходовых приложения для Интернета, электронная почта и веб-броузеры, основываются почти исключительно на сокетах TCP.

Вы тоже интенсивно используете UDP, не зная об этом. Всякий раз, когда ваша машина пытается найти сайт в Интернете, она посылает пакеты UDP вашему серверу DNS, запрашивая фактический адрес IP. Вы и сами можете использовать UDP, если захотите отправлять и принимать датаграммы. Датаграммы обходятся дешевле, чем соединения TCP, именно потому, что они не ориентированы на соединение; это значит, что они меньше похожи на выполнение телефонного звонка, чем на опускание письма в почтовый ящик. Однако в UDP нет той надежности, которую предоставляет TCP, из-за чего UDP более пригоден в ситуациях, когда нам неважно, что один-два пакета будут помяты, проколоты или разорваны. Или когда известно, что некоторый протокол более высокого уровня обеспечит какую-то меру избыточности или амортизации отказов (что и осуществляет DNS.)

Есть и другие возможности, но к ним обращаются значительно реже. Можно применять сокеты домена Unix, но они действуют только для локальной связи. Различные системы поддерживают другие протоколы, не использую-

щие IP. Несомненно, что где-нибудь кому-нибудь они интересны, но мы воздержимся от разговора о них.

Функции Perl, работающие с сокетами, имеют такие же имена, как соответствующие системные вызовы в C, но их аргументы обычно другие по двум причинам: во-первых, указатели файлов Perl действуют не так, как дескрипторы файлов C; и, во-вторых, Perl уже знает длину своих строк, поэтому такую информацию передавать не нужно. Подробные сведения обо всех системных вызовах, касающихся сокетов, можно найти в главе 29 «Функции».

Одна из проблем со старым кодом на Perl для сокетов заключалась в том, что использовались жестко закодированные значения констант, передаваемых в функции, работающие с сокетами, что разрушает переносимость. Как и большинство системных вызовов, те, которые относятся к сокетам, тихо, но аккуратно возвращают `undef` при неудаче выполнения, а не порождают исключительную ситуацию. Поэтому важно проверять значения, возвращаемые этими функциями, поскольку если им передается мусор, они начинают очень сильно шуметь. Увидев код, который делает что-то вроде явной установки `$AF_INET = 2`, будьте уверены, что вас ждут большие неприятности. Неизмеримо выше стоит подход, при котором используется модуль `Socket` или еще более дружественный модуль `IO::Socket`, – оба стандартные. Эти модули предоставляют различные константы и вспомогательные функции, которые потребуются для настройки клиентов и серверов. Наибольшего успеха программы, работающие с сокетами, добьются, если всегда будут начинаться следующим образом (и не забудьте ключ проверки меченых данных `-T` для серверов):

```
#!/usr/bin/perl -w
use strict;
use sigtrap;
use Socket;  # или IO::Socket
```

Как отмечается всюду, в значительной мере системное поведение Perl зависит от установленных библиотек C, и не все системы поддерживают все разновидности сокетов. Безопаснее всего, видимо, придерживаться обычных операций через сокеты с TCP и UDP. Например, если требуется, чтобы код имел шансы быть перенесенным в системы, о которых никто не думал при его разработке, не надо рассчитывать, что в них будет присутствовать поддержка надежного протокола упорядоченных пакетов. Также не следует передавать указатели открытых файлов между неродственными процессами через локальный сокет домена Unix. (Конечно, на многих Unix-машинах это делать можно – посмотрите страницу руководства по *recvmsg*(2).)

Если надо просто использовать стандартный сервис Интернета, например, электронную почту, телеконференции, службу доменных имен, FTP, Telnet, Web и т. д., то вместо того, чтобы начинать все с чистого листа, попробуйте использовать для этого модули CPAN. В число готовых модулей для этих целей входят `Net::SMTP` (или `Mail::Mailer`), `Net::NNTP`, `Net::DNS`, `Net::FTP`, `Net::Telnet` и различные модули, связанные с HTTP. Каждый из наборов модулей

libnet и libwww содержит много отдельных модулей для работы в сети. Области модулей в CPAN, которые вам могут понадобиться, включают в себя раздел 5 по сетевому взаимодействию и IPC, раздел 15 по модулям, относящимся к WWW, и раздел 16 по утилитам серверов и демонов.

В последующих разделах мы представляем несколько примеров клиентов и серверов, не слишком подробно объясняя каждую функцию, т. к. это, в основном, дублировало бы описания, имеющиеся в главе 29 «Функции».

Сетевые клиенты

Используйте сокеты доменов Интернета, если вам нужна надежная связь клиент-сервер между потенциально различными машинами.

Для создания клиента TCP, который соединяется с каким-то сервером, обычно легче всего применить стандартный модуль IO::Socket::INET:

```
use IO::Socket::INET;

$socket = IO::Socket::INET->new(PeerAddr => $remote_host,
                                PeerPort => $remote_port,
                                Proto    => "tcp",
                                Type     => SOCK_STREAM)
    or die "Невозможно соединиться с $remote_host:$remote_port : $!\n";

# послать что-то через сокет,
print $socket "Почему ты мне больше не звонишь?\n";

# прочесть ответ удаленной машины,
$answer = <$socket>;

# и по окончании завершить соединение.
close($socket);
```

Если у нас есть имя хоста и номер порта для подключения, а для остальных полей мы готовы сохранить значения по умолчанию, достаточно сокращенной формы вызова:

```
$socket = IO::Socket::INET->new("www.yahoo.com:80")
    or die "Невозможно соединиться с портом 80 на yahoo: $!";
```

Чтобы соединиться с помощью базового модуля Socket:

```
use Socket;

# создать сокет
socket(Server, PF_INET, SOCK_STREAM, getprotobyname('tcp'));

# построить адрес удаленной машины
$internet_addr = inet_aton($remote_host)
    or die "Невозможно преобразовать $remote_host в адрес Интернета: $!\n";
$paddr = sockaddr_in($remote_port, $internet_addr);

# соединение
connect(Server, $paddr)
    or die "Невозможно соединиться с $remote_host:$remote_port: $!\n";
```

```
select((select(Server), $| = 1)[0]);  # активизировать буферизацию команд

# послать что-то через сокет
print Server " Почему ты мне больше не звонишь?\n";

# прочесть ответ удаленной машины
$answer = <Server>;

# по окончании завершить соединение
close(Server);
```

Если необходимо закрыть соединение только со своей стороны, чтобы на удаленном конце был получен конец файла, но сохранить возможность чтения данных, поступающих с сервера, используйте системный вызов `shutdown` для полузакрытия:

```
# больше не выводить данные на сервер
shutdown(Server, 1);    # константа Socket::SHUT_WR в v5.6
```

Сетевые серверы

В продолжение приведем соответствующий сервер. Он довольно просто реализуется с помощью стандартного класса `IO::Socket::INET`:

```
use IO::Socket::INET;

$server = IO::Socket::INET->new(LocalPort => $server_port,
                                Type      => SOCK_STREAM,
                                Reuse     => 1,
                                Listen    => 10 )   # или SOMAXCONN
    or die "Невозможно стать сервером tcp на порту $server_port: $!\n";

while ($client = $server->accept()) {
    # $client является новым соединением
}
close($server);
```

Можно написать это и с помощью модуля более низкого уровня `Socket`:

```
use Socket;

# создать сокет
socket(Server, PF_INET, SOCK_STREAM, getprotobyname('tcp'));

# чтобы можно было быстро перезапустить сервер
setsockopt(Server, SOL_SOCKET, SO_REUSEADDR, 1);

# построить адрес моего сокета
$my_addr = sockaddr_in($server_port, INADDR_ANY);
bind(Server, $my_addr)
    or die "Невозможно привязаться к порту $server_port: $!\n";

# организовать очередь для входящих соединений
listen(Server, SOMAXCONN)
    or die "Невозможно слушать на порту $server_port: $!\n";
```

```
# принимать и обрабатывать соединения
while (accept(Client, Server)) {
    # сделать что-то с новым соединением Client
}
close(Server);
```

Клиенту не требуется привязываться к какому-то адресу, а вот серверу это нужно. Мы задали его адрес как INADDR_ANY, и это значит, что клиенты могут соединяться через любой имеющийся сетевой интерфейс. Для того чтобы работать через конкретный интерфейс (например, через внешний интерфейс машины с шлюзом или брандмауэром), используйте реальный адрес этого интерфейса. (Клиенты тоже могут это делать, но им это редко требуется.)

Для того чтобы узнать, что за машина с вами соединилась, вызовите getpeer-name для соединения с клиентом. Она возвращает адрес IP, который придется самостоятельно перевести в имя (если получится):

```
use Socket;
$other_end = getpeername(Client)
    or die "Невозможно идентифицировать клиента: $!\n";
($port, $iaddr) = unpack_sockaddr_in($other_end);
$actual_ip = inet_ntoa($iaddr);
$claimed_hostname = gethostbyaddr($iaddr, AF_INET);
```

Здесь имеется тривиальная возможность обмана, поскольку владелец этого IP-адреса может так настроить свои обратные таблицы, что они будут говорить все что угодно. В качестве некоторой дополнительной меры надежности оттранслируйте результат в обратном направлении:

```
@name_lookup = gethostbyname($claimed_hostname)
    or die "Could not reverse $claimed_hostname: $!\n";
@resolved_ips = map { inet_ntoa($_) } @name_lookup[ 4 .. $#name_lookup ];
$might_spoof = !grep { $actual_ip eq $_ } @resolved_ips;
```

Когда клиент соединится с вашим сервером, сервер может осуществлять ввод/вывод через указатель этого клиента. Но поскольку сервер так занят, он не сможет обслуживать дальнейшие запросы, поступающие от других клиентов. Чтобы избежать замыкания на одновременной работе только с одним клиентом, многие серверы немедленно создают с помощью fork свои клоны для обработки каждого входящего соединения. (Другие выполняют ветвление заранее или мультиплексируют ввод-вывод между несколькими клиентами посредством системного вызова select.)

```
REQUEST:
while (accept(Client, Server)) {
    if ($kidpid = fork) {
        close Client;         # родитель закрывает неиспользуемый указатель
        next REQUEST;
    }
    defined($kidpid)   or die "невозможно выполнить fork: $!" ;
```

```
    close Server;             # потомок закрывает неиспользуемый указатель

    select(Client);           # новые значения по умолчанию  для вывода
    $| = 1;                   # автовыгрузка буфера

    # ввод-вывод через указатель Client кода дочернего процесса для каждого
    # соединения
    $input = <Client>;
    print Client "output\n";  # или STDOUT, то же самое

    open(STDIN, "<<&Client")    or die "невозможно дублировать клиента: $!";
    open(STDOUT, ">&Client")    or die "невозможно дублировать клиента: $!";
    open(STDERR, ">&Client")    or die "невозможно дублировать клиента: $!";

    # запустим калькулятор, только для примера,
    system("bc -l");     # или что вам угодно, лишь бы
                         # не было управляющих символов для оболочки!
    print "done\n";      # все еще клиенту

    close Client;
    exit;  # не пускать дочерний процесс назад к accept!
}
```

Для каждого поступившего запроса этот сервер клонирует дочерний процесс через `fork`. Благодаря этому он может одновременно обрабатывать много запросов, пока сохраняется возможность создавать новые процессы. (Возможно, вы захотите ее ограничить.) Даже если не выполнять `fork`, `listen` разрешает до `SOMAXCONN` (обычно пяти или более) ожидающих соединений. Каждое соединение использует некоторые ресурсы, хотя и в меньшем объеме, чем процесс. Разделяющиеся на процессы серверы должны позаботиться об уборке своих отработавших потомков (называемых «зомби» на жаргоне Unix), иначе они быстро заполнят всю таблицу процессов. За вас это может сделать код `REAPER`, обсуждавшийся в разделе «Сигналы», либо вы сами можете присвоить `$SIG{CHLD} = 'IGNORE'`.

Прежде чем выполнить новую команду, мы подключаем стандартные устройства ввода, вывода (и вывода ошибок) к соединению с клиентом. В результате любая команда, читающая из `STDIN` и пишущая в `STDOUT`, может общаться с удаленной машиной. Без переназначения команда не смогла бы найти указатель клиента, который по умолчанию закрывается при переходе через границу `exec`.

Если вы пишете сетевой сервер, мы настоятельно рекомендуем вам использовать ключ `-T` для активации проверки меченых данных, даже если вы не запускаете setuid или setgid. Это всегда полезно для серверов или любых других программ, выполняемых от чужого имени (как все сценарии CGI), поскольку это уменьшает шансы посторонних проникнуть в вашу систему. Подробнее обо всем этом см. в разделе «Обработка ненадежных данных» в главе 23 «Защита данных».

Одно дополнительное соображение, касающееся создания программ для Интернета: многие протоколы требуют, чтобы концом строки была комбинация `CRLF`, которую можно задать несколькими способами: `"\015\12"`, `"\xd\xa"`

или даже chr(13).chr(10). В версии Perl 5.6 выражение v13.10 дает такую же строку. (На многих машинах для обозначения CRLF годится последовательность "\r\n", но от нее следует отказаться, если необходимо обеспечить возможность переноса на Macintosh, где значения \r и \n являются противоположными!) Многие программы Интернета на практике применяют в качестве окончания строки просто "\012", но лишь потому, что программы Интернета обычно стараются быть либеральными в отношении того, что они принимают, и строгими в отношении того, что выдают. (Если бы только люди поступали так же...)

Передача сообщений

Как упоминалось выше, связь через UDP требует значительно меньших накладных расходов, но не обеспечивает надежности, поскольку ничто не гарантирует поступление сообщений в правильном порядке (и даже вообще их поступление). Часто говорят, что UDP означает Unreliable Datagram Protocol – ненадежный протокол датаграмм.

Тем не менее, UDP имеет некоторые преимущества перед TCP, в том числе возможность широковещания или одновременной передачи сразу для группы хостов (обычно в локальной сети). Если надежность стала предметом вашей особой заботы и вы стали встраивать проверки в свою систему передачи сообщений, то, возможно, следует предпочесть TCP. Правда, расходы на установление и разрыв соединения TCP будут больше, но если удастся амортизировать их передачей большого числа сообщений (или одного длинного сообщения), то это обстоятельство не очень существенно.

Как бы то ни было, вот пример программы, использующей UDP. Она связывается с портом UDP времени на машинах, указанных в командной строке, или на всех, которые найдет с помощью широковещательного адреса, если аргументы не указаны.[1] Не на всех машинах включен сервер времени, особенно за границами брандмауэра, но те, которые смогут, возвратят вам 4-байтовое целое число, порядок байтов в котором соответствует сетевому формату, представляющее собой текущее время по мнению этой машины. Время возвращается в виде числа секунд, прошедших с 1900 года. Из этого числа нужно вычесть число секунд между 1900 и 1970 годами, чтобы передать это время функциям преобразования localtime или gmtime.

```
#!/usr/bin/perl
# clockdrift - сравнить часы других систем с этими
#              без аргументов, передача всем, кто слушает.
#              ждать ответа полсекунды.

use v5.6.0;  # или выше
use warnings;
use strict;
```

[1] Если он не действует, запустите *ifconfig -a* и найдите правильный локальный широковещательный адрес.

```
use Socket;

unshift(@ARGV, inet_ntoa(INADDR_BROADCAST))
    unless @ARGV;

socket(my $msgsock, PF_INET, SOCK_DGRAM, getprotobyname("udp"))
    or die "сокет: $!";

# Требуется некоторым "неиспарвным" (borked)[1] машинам и не должно повредить
остальным.
setsockopt($msgsock, SOL_SOCKET, SO_BROADCAST, 1)
    or die "setsockopt: $!";

my $portno = getservbyname("time", "udp")
    or die "нет порта времени udp";

for my $target (@ARGV) {
    print "Отправка $target:$portno\n";
    my $destpaddr = sockaddr_in($portno, inet_aton($target));
    send($msgsock, "x", 0, $destpaddr)
        or die "send: $!";
}

# сервис времени возвращает 32-разрядное число секунд после 1900
my $FROM_1900_TO_EPOCH = 2_208_988_800;
my $time_fmt = "N";   # и так в этом двоичном формате
my $time_len = length(pack($time_fmt, 1));  # годится любое число

my $inmask = '';  # строка для хранения битов fileno для select
vec($inmask, fileno($msgsock), 1) = 1;

# ждать полсекунды появления входных данных
while (select(my $outmask = $inmask, undef, undef, 0.5)) {
    defined(my $srcpaddr = recv($msgsock, my $bintime, $time_len, 0))
        or die "recv: $!";
    my($port, $ipaddr) = sockaddr_in($srcpaddr);
    my $sendhost = sprintf "%s [%s]",
                    gethostbyaddr($ipaddr, AF_INET) || 'UNKNOWN',
                    inet_ntoa($ipaddr);
    my $delta = unpack($time_fmt, $bintime) -
                      $FROM_1900_TO_EPOCH - time();
    print "Часы на $sendhost спешат на $delta секунд относительно данных.\n";
```

[1] Borked – эквивалент жаргонного слова «borken» – представляющего собой намеренно искажаемое хакерами «broken». – *Примеч. ред.*

17

Потоки

Параллельное программирование значительно сложнее, чем может показаться. Представьте себе, что вы берете рецепт из поваренной книги и модифицируете его так, чтобы несколько десятков поваров могли работать с ним в одно и то же время. Можно выбрать два подхода.

Первый способ состоит в том, чтобы предоставить каждому повару собственную кухню, обеспеченную собственными исходными продуктами и посудой. Для рецептов, которые легко могут быть разделены на части, и пищевых продуктов, которые могут легко перемещаться из кухни в кухню, этот метод работает хорошо, потому что повара не допускаются в чужие кухни.

В другом случае можно поместить всех поваров в одну кухню и дать им возможность самим разбираться с тем, например, кому и когда использовать миксер. Это может привести к беспорядку, особенно если начнут летать ножи для резки мяса.

Эти два подхода соответствуют двум моделям параллельного программирования на компьютерах. Первый – модель мультиобработки, обычная для традиционных систем Unix, в которых каждый поток управления имеет собственный набор ресурсов, совокупность которых мы называем процессом. Вторая модель – многопоточная, в соответствии с которой каждый поток имеет общие ресурсы со всеми другими потоками управления. Или не имеет, как иногда может быть (или должно быть).

Все мы знаем, что повара любят командовать, и это хорошо, потому что повара должны командовать, чтобы выполнить то, чего мы от них хотим. Но повара должны быть тем или иным образом организованы.

Perl поддерживает обе модели организации. В этой главе мы будем называть их *моделью процессов* (*process model*) и *моделью потоков* (*thread model*).

Модель процессов

Мы не будем здесь подробно обсуждать модель процессов просто потому, что она пронизывает оставшуюся часть этой книги. Perl родился на Unix-системах, поэтому он вобрал в себя понятие о том, что каждый процесс занимается своим делом. Если процесс хочет начать некоторую параллельную обработку, то по логике вещей он должен открыть параллельный процесс, т. е. должно произойти ответвление полновесного нового процесса, который по умолчанию мало чем пользуется совместно с родительским процессом, за исключением некоторых дескрипторов файлов. (Может показаться, что значительно бо́льшая часть ресурсов разделяется между родителем и потомком, но в основном состояние родительского процесса просто дублируется в процессе потомка, а не используется совместно в подлинном смысле. Операционная система может, конечно, проявить лень в осуществлении этого логического разделения, и тогда мы называем это семантикой «копирования при записи» (copy-on-write), но мы вообще не делали бы копии, если бы вначале не было логического разделения.)

Исторически промышленный взгляд на мультиобработку выявил некоторые проблемы на системах Microsoft, поскольку в Windows модель мультиобработки была проработана недостаточно (а на ту, что есть, она при параллельном программировании часто не полагается). Вместо нее обычно использовался многопоточный подход.

Однако героическими усилиями в версии Perl 5.6 теперь реализована операция `fork` под Windows путем клонирования нового объекта интерпретатора в рамках того же самого процесса. Это означает, что большинство примеров в оставшейся части книги, в которых используется `fork`, теперь будут работать в Windows. Клонированный интерпретатор использует тот же программный код, что и остальные интерпретаторы, но получает для работы собственный экземпляр данных. (Конечно, могут сохраниться проблемы с библиотеками C, которые не умеют работать с потоками.)

Этот подход к мультиобработке окрестили *ithreads*, сокращенно от «interpreter threads», что означает «потоки интерпретаторов». Начальным импульсом к его реализации послужило желание эмулировать `fork` на системах Microsoft. Однако вскоре стало понятно, что хотя другие интерпретаторы выполняются как отдельные потоки, это происходит в рамках одного процесса, поэтому легко организовать их совместный доступ к данным, даже если по умолчанию он отсутствует.

Это противоположно обычной модели потоков, в которой по умолчанию все является общим, и нужно принимать меры, чтобы *не* осуществлялся совместный доступ к чему-либо. Не следует, однако, рассматривать эти две модели как совершенно различные, поскольку и та и другая наводят мост

через одну и ту же реку, только с противоположных берегов. Фактическое решение любой задачи параллельной обработки в какой-то мере предполагает как готовность делиться, так и некоторый эгоизм.

В долгосрочные планы входит намерение расширить модель потоков интерпретаторов так, чтобы она допускала необходимую или желательную меру совместного доступа. На момент написания этой книги единственным доступным пользователю интерфейсом для потоков интерпретаторов являлся вызов `fork` в версиях Perl для Microsoft. Мы полагаем, что в итоге этот подход приведет к созданию программ более высокого качества, чем при стандартном применении потоков. В сущности, проще управлять такой экономикой, в которой каждый владеет своей собственностью, нежели такой, в которой каждый владеет всем. Нельзя сказать, что в капиталистической экономике никто ни с кем не делится, а при коммунизме все разбазаривается. Ситуация стремится к чему-то среднему. Случается и социализм. Но в больших группах людей общая по умолчанию собственность действенна только тогда, когда есть «шеф-повар» с большим мясным ножом, считающий, что ему принадлежит все.

Конечно, реальное управление любым компьютером осуществляет фашистский диктатор по имени «операционная система». Но мудрый диктатор знает, когда позволить людям считать себя капиталистами, а когда – коммунистами.

Модель потоков

Потоковая модель мультиобработки была впервые введена в Perl в версии 5.005 как экспериментальная функция. (Под «потоковой моделью» мы подразумеваем потоки, которые по умолчанию имеют совместный доступ к ресурсам данных, а не новую модель потоков интерпретаторов из версии 5.6.) В некоторых отношениях эта модель потоков остается экспериментальной даже в версии 5.6, поскольку Perl – язык богатый, а многопоточность может создать неразбериху даже в самом простом языке. В семантике Perl сохраняются укромные места, которые не очень хорошо согласуются с представлениями о том, что все общее. Новая модель потоков интерпретаторов представляет собой попытку обойти эти проблемы, и в какой-то момент в будущем текущая модель потоков, возможно, будет включена в модель потоков интерпретаторов (когда мы получим интерфейс к потокам интерпретаторов, который позволит задать совместный доступ по умолчанию ко всему, что только можно). Однако несмотря на свои недостатки, текущая «экспериментальная» модель продолжает быть полезной во многих практических ситуациях, в которых единственная альтернатива роли подопытного кролика еще менее желательна. При помощи потоков в Perl можно создавать достаточно надежные приложения, но необходимо соблюдать большую осторожность. Следует хотя бы рассмотреть возможность использования `fork`, если можно придумать, как решить задачу с помощью каналов, а не структур данных с общим доступом.

Однако некоторые алгоритмы выражаются легче, когда несколько задач получают простой и эффективный доступ к одному и тому же пулу данных.[1] Это способствует написанию более коротких и простых программ. А поскольку ядру ОС при создании потоков не приходится копировать таблицы страниц данных (даже при копировании по записи), запуск задач должен осуществляться быстрее. Аналогично, переключение контекста может происходить быстрее, если ядру не нужно осуществлять замену таблиц страниц. (На практике в потоках уровня пользователя ядро вообще не участвует, впрочем, конечно, проблемы, связанные с потоками уровня пользователя, отсутствуют в потоках ядра.)

Это то, что относится к положительным сторонам. Теперь отметим и некоторые отрицательные. Уже упоминалось, что поддержка потоков в Perl является отчасти экспериментальной. Но в любом случае программирование с использованием потоков остается рискованным занятием. Способность одного потока выполнения вольно или невольно проделывать бреши в пространстве данных другого потока создает больше возможностей для краха системы, чем можно себе вообразить. Может показаться, что это легко исправить, поставив блокировки на все данные с совместным доступом. Действительно, блокировка данных для совместного доступа необходима, но создать корректные протоколы блокировки очень трудно, причем ошибки приводят к взаимной блокировке или недетерминированным результатам. Если в вашей программе возникают трудности с синхронизацией, то применение потоков не только усугубит их, но и затруднит обнаружение.

Программист несет ответственность не только за порядок в собственных данных совместного доступа, но и в данных модулей Perl и библиотек C, обращение к которым он осуществляет. Код на Perl может использовать потоки, защищенные на 100%, но, допустив вызов модуля или подпрограммы C без защиты потоков, вы прогорите. Следует исходить из того, что в любом модуле потоки не являются защищенными, если не доказано обратное. Это относится даже к некоторым стандартным модулям, а возможно, и к большей их части.

Мы еще не привели вас в уныние? Нет? Тогда заметим, что вы в значительной мере зависите от особенностей реализации политики планирования и вытеснения в библиотеке поддержки потоков, имеющейся в вашей операционной системе. Некоторые библиотеки осуществляют переключение потоков только при блокирующих системных вызовах, другие – только по истечении кванта времени (процесса или потока), а третьи – только при явном указании.

Да, между прочим: если процесс получает сигнал, то поток, которому он будет передан, полностью определяется системой.

Чтобы программировать потоки в Perl, нужно скомпилировать специальную его версию, следуя указаниям из файла *README.threads* в каталоге

---

[1] Модель памяти общего доступа в System V, обсуждаемую в последней главе, нельзя с полным основанием охарактеризовать как «простую и эффективную».

с исходным кодом Perl. Можно не сомневаться, что эта версия Perl будет работать несколько медленнее, чем стандартный исполняемый модуль Perl.

Не следует думать, что если вы знакомы с программированием потоков в других моделях (POSIX, DEC, Microsoft и др.), то этого достаточно, чтобы знать, как работают потоки в Perl. Как и в других случаях, Perl – это Perl, а не C++, Java или что-либо еще. Например, в нем нет приоритетов потоков реального времени (и способа обойти их отсутствие). Нет также взаимоисключений (mutexes). Используйте обычную блокировку, модуль `Thread::Semaphore` или средства `cond_wait`.

Еще не впали в уныние? Очень хорошо, потому что потоки – это действительно сильно. Впереди вас ждет много «развлечений».

Модуль Thread

В настоящее время интерфейс для потоков в Perl определяется в модуле Thread. Кроме того, добавлено еще одно ключевое слово Perl, оператор `lock`. Мы поговорим о `lock` далее в этой главе. Остальные стандартные модули потоков основываются на этом базовом интерфейсе.

Модуль `Thread` предоставляет следующие методы класса:

Метод	Использование
`new`	Создает новый поток `Thread`
`self`	Возвращает текущий объект `Thread`
`list`	Возвращает список объектов `Thread`

У объектов `Thread` есть следующие методы объекта:

Метод	Использование
`Join`	Объединение потока (размножение ошибок)
`Eval`	Объединение потока (перехват ошибок)
`equal`	Проверка идентичности двух потоков
`Tid`	Возврат внутреннего ID потока

Кроме того, модуль `Thread` предоставляет следующие две импортируемые функции:

Функция	Использование
`yield`	Указание планировщику о запуске другого потока
`async`	Создать `Thread` через замыкание
`cond_signal`	Разбудить ровно один поток, ожидающий по переменной по `cond_wait()`
`cond_broadcast`	Разбудить все потоки, ожидающие по переменной по `cond_wait()`
`cond_wait`	Ждать по переменной до пробуждения посредством `cond_signal()` или `cond_broadcast()` с этой переменной

Создание потоков

Породить поток можно одним из двух способов: с помощью метода класса Thread->new или с помощью функции async. В том и другом случае возвращается объект Thread. Thread->new принимает ссылку на функцию, которая должна быть вызвана, и аргументы, которые должны быть ей переданы:

```
use Thread;
...
$t = Thread->new( \&func, $arg1, $arg2);
```

Часто требуется передать только первый аргумент – замыкание:

```
my $something;
$t = Thread->new( sub { say($something) } );
```

В этом особом случае функция async позволяет упростить запись («подсластить» синтаксис):

```
use Thread qw(async);
...
my $something;
$t = async {
    say($something);
};
```

Обратите внимание, что функция async импортирована явным образом. Можно, конечно, использовать и полностью квалифицированное имя Thread::async, но тогда синтаксический «сахар» не столь сладок. Поскольку async принимает только замыкание, все, что можно ей передать, это лексическая переменная, находящаяся в области видимости в данный момент.

Уничтожение потоков

Поток, будучи запущен и став жертвой капризов библиотеки поддержки потоков, продолжает самостоятельное выполнение до того момента, пока не произойдет завершение функции верхнего уровня – той, которая была передана конструктору. Если требуется досрочно завершить поток, нужно осуществить возврат из этой функции верхнего уровня.[1]

Пусть все готово для возврата из вашей подпрограммы верхнего уровня, но *куда* он будет производиться? Поток, которым порожден данный, вероятно занят другими делами и уже не ждет ответа от вызванного метода. Ответ достаточно прост: поток ждет, пока где-нибудь не будет вызван метод, действительно ожидающий возвращаемого значения. Этот метод называется join, поскольку по замыслу объединяет два потока снова в один:

```
$retval = $t->join();    #  объединение потока $t
```

[1] Не вызывайте exit! Он попытается завершить весь ваш процесс и, вероятно, преуспеет в этом. Но завершения процесса не произойдет, пока не завершатся все потоки, а некоторые из них могут отказаться завершаться по exit. Подробнее об этом рассказано далее.

Операция join напоминает waitpid для дочернего процесса. Если поток уже завершен, метод join возвращается немедленно со значением, возвращенным подпрограммой верхнего уровня потока. Если поток еще жив, join действует как блокирующий вызов, прерывающий выполнение вызывающего потока на неопределенное время (возможности тайм-аута нет). Когда поток, в конце концов, завершается, происходит возврат из join.

Однако, в отличие от waitpid, работающего только с потомками данного процесса, с помощью join каждый поток может соединяться с любым другим потоком процесса. Это означает, что осуществляющий соединение поток не обязательно должен быть главным или родительским. Единственное ограничение состоит в том, что поток не должен соединяться с самим собой (это напоминало бы распорядительство на собственных похоронах) и соединяться с потоком, с которым уже произведено соединение (а это было бы похоже на драку двух распорядителей похорон за тело). При попытке совершить то или другое будет возбуждена исключительная ситуация.

Возвращаемое join значение не обязательно должно быть скаляром, это может быть и список:

```
use Thread 'async';

$t1 = async {
    my @stuff = getpwuid($>);
    return @stuff;
};

$t2 = async {
    my $motd = `cat /etc/motd`;
    return $motd;
};

@retlist = $t1->join();
$retval  = $t2->join();

print "1st kid returned @retlist\n";
print "2nd kid returned $retval\n";
```

На самом деле выражение, возвращаемое потоком, всегда вычисляется в списковом контексте, даже если join вызывается в скалярном контексте; в этом случае возвращается последнее значение списка.

Перехват исключительных ситуаций, возбуждаемых join

Если поток завершается, создавая не перехватываемую исключительную ситуацию, это не обязательно сразу убивает всю программу. Это было бы нехорошо. Вместо этого, когда для потока выполняется join, исключительную ситуацию создает сам join. Использование join с потоком указывает на готовность к дальнейшему распространению исключительных ситуаций, создаваемых потоком. Если, напротив, вы хотите сразу перехватить исключительную ситуацию, обратитесь к методу eval, который, как и его встроенный аналог, помещает исключительную ситуацию в $@:

```
$retval = $t->eval();   # перехватить ошибки join
if ($@) {
    warn " аварийная ситуация в потоке: $@";
}
else {
    print " поток возвратил $retval\n";
}
```

Хотя это правило не является обязательным, можно придерживаться политики присоединения к потоку только из потока, создавшего тот, к которому происходит присоединение. Иными словами, дочерние потоки собираются только из родительских, их породивших. Благодаря этому несколько легче определить, где и какие исключительные ситуации должны обрабатываться.

Метод detach

В качестве альтернативного способа закрытия потока, если вы не собираетесь в дальнейшем объединяться с ним, чтобы получить возвращаемое значение, можно вызвать для него метод `detach`, и Perl завершит его работу сам. Присоединиться к потоку будет уже невозможно. Это несколько похоже на наследование процессу программой *init* в Unix, за исключением того, что под Unix родительский процесс должен завершиться.

Метод `detach` не переводит поток в «фоновый режим»: если попытаться завершить программу, когда поток, запущенный через `detach`, все еще выполняется, exit зависнет в ожидании завершения потока. Метод `detach` лишь освобождает от необходимости «зачистки». Он просто сообщает Perl, что не нужно хранить возвращаемое значение и статус завершения после окончания выполнения потока. В некотором смысле, `detach` указывает Perl, что нужно выполнить неявный вызов `join` при завершении потока и отбросить результаты выполнения. Важно обратить внимание на то, что если с потоком не выполняются ни `join`, ни `detach`, и при этом поток возвращает какой-нибудь большой список, занимаемая им память будет потеряна до самого конца, поскольку Perl будет держать ее в слабой надежде (в данном случае весьма слабой), что кому-то потребуется выполнить `join` для этого потока.

Кроме того, исключительные ситуации, создающиеся в потоках, вызванных с помощью `detach`, не распространяются дальше через `join`, поскольку их быть не может. Используйте разумным образом `eval {}` в функции верхнего уровня и сообщайте об ошибках другими способами.

Идентификация потоков

У каждого потока Perl есть отличительный идентификационный номер, который можно получить с помощью метода объекта `tid`:

```
$his_tidno = $t1->tid();
```

Поток может получить доступ к своему объекту потока, вызвав `Thread->self`. Не надо смешивать это с ID потока: свой ID поток получает так:

```
$mytid = Thread->self->tid();   # $$ for threads, as it were.
```

Сравнить между собой объекты потоков можно одним из следующих способов:

```
Thread::equal($t1, $t2)
$t1->equal($t2)
$t1->tid() == $td->tid()
```

Перечисление текущих потоков

Получить список объектов текущих потоков для текущего процесса можно с помощью метода класса `Thread->list`. Список включает в себя выполняющиеся потоки и те завершенные, с которыми еще не было соединения. Получить список можно, находясь в любом потоке.

```
for my $t (Thread->list()) {
    printf "$t has tid = %d\n", $t->tid();
}
```

Передача процессорного времени

В модуле `Thread` поддерживается импортируемая функция с именем `yield`. Ее назначение в том, чтобы поток отдавал управление процессором. К сожалению, механизм происходящего полностью зависит от особенностей реализации потоков в системе. Тем не менее, считается хорошим тоном время от времени отдавать контроль над CPU:

```
use Thread 'yield';
yield();
```

Скобки здесь не обязательны. Это даже надежнее с синтаксической точки зрения, поскольку перехватывает неизменную опечатку «yeild»:

```
use strict;
use Thread 'yield';
yeild;          # Компилятор стонет.
yield;          # Ok.
```

Доступ к данным

То, что мы до сих пор рассматривали, на самом деле не очень сложно, но сейчас мы исправим это упущение. Все, чем мы занимались, никак не затрагивало параллельную природу выполнения потоков. Доступ к совместно используемым данным в корне меняет ситуацию.

Код, манипулирующий потоками, имеет дело с такими же ограничениями на видимость данных, как и любой другой код в Perl. Глобальные объекты доступны через таблицы глобальных символов, а лексические – через какую-либо содержащую их область лексической видимости (временную память).

Однако факт существования в программе нескольких потоков управления становится решающим аргументом в работе. Два потока не допускаются одновременно к одной и той же глобальной переменной, иначе они могут «затоптать» друг друга. (Чем это закончится, зависит от характера доступа.)

Аналогично, двум потокам нельзя разрешить одновременный доступ к одной и той же лексической переменной, поскольку лексические переменные ведут себя подобно глобальным, если объявлены вне областей видимости замыканий, используемых потоками. Запуск потоков через ссылки на подпрограммы (с помощью `Thread->new`), а не через замыкания (с помощью `async`), помогает ограничить доступ к лексическим переменным, если это то, что вам требуется. (Иногда, однако, требуется другое.)

Perl решает проблему для некоторых специальных встроенных переменных типа `$!` или `$_`, или `@_`, делая содержащиеся в них данные специфическими для потока. Плохо, однако, что основные переменные, содержащиеся в обычных пакетах, не защищены от вмешательства других потоков.

К счастью, обычно не приходится беспокоиться о лексических переменных, если они объявлены внутри текущего потока, поскольку каждый поток создает при запуске собственную область лексической видимости, отдельную от других потоков. О лексических переменных нужно беспокоиться только тогда, когда они совместно используются потоками, например, при передаче ссылок или при ссылке на лексические переменные из замыканий, выполняющихся несколькими потоками.

Синхронизация доступа с помощью блокировок

Если к одному и тому же объекту в одно и то же время имеет доступ более чем один агент, то происходят конфликты – как на перекрестке дорог. Единственной защитой является тщательная организация блокировки.

Встроенная функция блокировки (`lock`) служит в Perl механизмом включения красного/зеленого света для управления доступом. Хотя слово `lock` является разновидностью ключевого, оно застенчиво по характеру – в том смысле, что встроенная функция *не* используется, если компилятор уже обнаружил в коде пользователя определение `sub lock {}`. Это сделано для обеспечения обратной совместимости в будущем. Однако `CORE::lock` всегда является встроенной. (В *perl*, скомпилированном без поддержки потоков, вызов `lock` не является ошибкой; это безобидная функция, не производящая никаких действий (no-op), по крайней мере, в последних версиях).

Подобно тому как оператор `flock` блокирует только другие экземпляры `flock`, но не фактический ввод/вывод, так и оператор `lock` блокирует только другие экземпляры `lock`, а не обычный доступ к данным. Это, в сущности, *рекомендованная* блокировка, аналогично сигналу светофора.[1]

Можно блокировать как отдельные скалярные переменные, так и целые массивы и хеши.

---

[1] Некоторые сигналы на железнодорожных перекрестках являются обязательными (если они оборудованы шлагбаумами), и кое-кто считает, что блокировки тоже должны быть обязательными. Но представьте себе картину, когда на каждом перекрестке будет стоять шлагбаум, поднимающийся или опускающийся при каждой смене сигнала светофора!

```
lock $var;
lock @values;
lock %table;
```

Однако блокировка составного объекта не влечет неявной блокировки всех его скалярных составляющих:

```
lock @values;       # в потоке 1
...
lock $values[23];   # в потоке 2 - - не заблокирует!
```

Если блокировка применяется к ссылке, то автоматически блокируется объект ссылки, т. е. одно разыменование вы получаете бесплатно.[1] Это удобно, потому что объекты всегда находятся за ссылками, а часто требуется блокировать объекты. (И почти никогда не требуется блокировать сами ссылки.)

Проблема со светофором, конечно, в том, что половину времени приходится простаивать перед красным светом. Аналогично, `lock` представляет собой блокирующий вызов: поток зависает, пока не будет снята блокировка. Средства завершения по тайм-ауту нет. Средства снятия блокировки тоже нет, поскольку область видимости блокировок динамическая. Блокировка длится до конца осуществивших ее блока, файла или `eval`. При выходе из области видимости блокировки автоматически снимаются.

Блокировки являются также рекурсивными. Это означает, что если в функции блокируется переменная и происходит рекурсивный вызов функции при сохранении блокировки, та же переменная может быть заблокирована в том же потоке снова. Блокировка окончательно снимается после выхода из всех стеков, владеющих блокировками.

Вот простой пример, показывающий, что может произойти в отсутствие блокировки. Мы используем принудительное переключение контекста с помощью `yield`, чтобы показать также, какие проблемы могут возникать при вытесняющем планировании (preemptive scheduling):

```
use Thread qw/async yield/;
my $var = 0;
sub abump {
    if ($var == 0) {
        yield;
        $var++;
    }
}

my $t1 = new Thread \&abump;
my $t2 = new Thread \&abump;

for my $t ($t1, $t2) { $t->join }
print "var is $var\n";
```

[1] В том смысле, что еще одну операцию Perl выполняет автоматически.

Эта программа всегда (в некотором смысле) выводит 2, поскольку мы решили инкрементировать переменную, увидев, что ее значение 0, но прежде чем мы успели сделать это, то же самое решил сделать другой поток.

Этот конфликт можно предотвратить, тривиально добавив блокировку до проверки значения $var. Следующий код всегда выводит 1:

```
sub abump {
    lock $var;
    if ($var == 0) {
        yield;
        $var++;
    }
}
```

Запомните, что явной функции unlock нет. Чтобы управлять разблокированием, просто добавьте еще одну, вложенную, область видимости, чтобы блокировка была снята при выходе из нее:

```
sub abump {
    {
        lock $var;
        if ($var == 0) {
            yield;
            $var++;
        }
    }  # здесь блокировка снята!
    # другой код с незаблокированной $var
}
```

Взаимоблокировка

Взаимоблокировка – это бич для тех, кто программирует потоки, поскольку ее легко вызвать по неосторожности и трудно избежать, даже если очень стараться. Вот простой пример взаимоблокировки:

```
my $t1 = async {
    lock $a; yield; lock $b;
    $a++; $b++
};
my $t2 = async {
    lock $b; yield; lock $a;
    $b++; $a++
};
```

Решение проблемы в том, чтобы все стороны, использующие некоторый набор блокировок, осуществляли их в одном и том же порядке.

Неплохо также минимизировать продолжительность удержания блокировок. (По крайней мере это полезно для повышения быстродействия. Но если делать это для уменьшения риска взаимоблокировки, то можно лишь затруднить воспроизведение и диагностирование ошибки.)

Блокировка подпрограмм

Можно блокировать и подпрограммы:

```
lock &func;
```

В отличие от блокировок данных, являющихся лишь рекомендательными, блокировки подпрограмм *обязательны*. Вход в подпрограмму разрешается только потоку, осуществившему блокировку.

Рассмотрим следующий код, содержащий условие гонок по переменной `$done`. (Операторы `yield` служат только для большей наглядности.)

```
use Thread qw/async yield/;
my $done = 0;
sub frob {
    my $arg = shift;
    my $tid = Thread->self->tid;
    print "thread $tid: frob $arg\n";
    yield;
    unless ($done) {
        yield;
        $done++;
        frob($arg + 10);
    }
}
```

Если запустить его таким образом:

```
my @t;
for my $i (1..3) {
    push @t, Thread->new(\&frob, $i);
}
for (@t) { $_->join }
print "done is $done\n";
```

то выведено будет следующее (может быть выведено – содержание не детерминировано):

```
thread 1: frob 1
thread 2: frob 2
thread 3: frob 3
thread 1: frob 11
thread 2: frob 12
thread 3: frob 13
done is 3
```

Однако если запустить его таким образом:

```
for my $i (1..3) {
    push @t, async {
        lock &frob;
        frob($i);
```

```
    };
}
for (@t) { $_->join }
print "done is $done\n";
```

то выдача будет следующей:

```
thread 1: frob 1
thread 1: frob 11
thread 2: frob 2
thread 3: frob 3
done is 1
```

Атрибут locked

Подчинение блокировкам подпрограмм является обязательным, но, прежде всего, ничто никого не обязывает устанавливать на них блокировки. Можно сказать, что установка блокировки является рекомендательной. Но для некоторых подпрограмм бывает желательным, чтобы они могли потребовать установки на них блокировки перед вызовом.

Эту задачу решает атрибут подпрограммы `locked`. Он действует быстрее, чем вызов `lock &sub`, поскольку о блокировке становится известно во время компиляции, а не выполнения. Однако режим выполнения оказывается таким же, как при явной блокировке, продемонстрированной выше. Синтаксис имеет следующий вид:

```
sub frob : locked {
    # как ранее
}
```

Если у функции есть прототип, он помещается между именем и атрибутами:

```
sub frob ($) : locked {
    # как ранее
}
```

Методы блокировки

Автоматическая блокировка подпрограммы – это здорово, но иногда избыточно. При вызове метода объекта обычно не существенно, выполняется ли одновременно несколько методов, если они относятся к разным объектам. Поэтому на самом деле бывает нужно заблокировать объект, к которому применяется метод. Это делается путем добавления атрибута `method` к определению подпрограммы:

```
sub frob : locked method {
    # как ранее
}
```

При обращении к нему как к методу вызывающий объект блокируется, и к нему разрешается лишь последовательный доступ, однако тот же метод

может применяться к другим объектам. Если метод вызывается не с объектом, атрибут все же старается проявить себя правильным образом: при вызове блокированного метода как метода класса (`Package->new`, а не `$obj->new`) блокируется таблица символов класса. При вызове заблокированного метода как обычной процедуры Perl создает исключительную ситуацию.

Переменные условий

Переменная условия (condition variable) позволяет потоку отказаться от процессорного времени, пока не будет удовлетворен некоторый критерий. Переменные условия предназначены служить точками координации выполнения потоков, если нужна более высокая степень контроля, чем предоставляемая простыми блокировками. С другой стороны, издержки должны быть не выше, чем в случае применения блокировок, и в переменных условия это учтено. Просто используются обычные блокировки и обычные условные операторы. Если условие не выполняется, приходится принимать чрезвычайные меры посредством функции `cond_wait`, но мы оптимизируем программу в расчете на успех, поскольку в правильно спроектированном приложении текущее состояние не должно быть узким местом.

Помимо блокировки и проверки в число основных операций с переменными условия входят отправка и получение «сигнальных» событий (не реальных сигналов в смысле `%SIG`). Вы либо прерываете выполнение своего потока и ждете получения события, либо посылаете событие, чтобы «разбудить» потоки, ожидающие выполнения некоторого условия. В модуле Thread есть три импортируемые функции для осуществления этого: `cond_wait`, `cond_signal` и `cond_broadcast`. Это примитивные механизмы, положенные в основу более абстрактных модулей типа `Thread::Queue` и `Thread::Semaphore`. Часто удобнее использовать предоставляемые ими абстракции, если это возможно.

Функция `cond_wait` берет переменную, уже заблокированную в текущем потоке, разблокирует ее, а затем осуществляет блокировку, пока другой поток не выполнит `cond_signal` или `cond_broadcast` для той же заблокированной переменной.

Переменная, блокированная `cond_wait`, снова блокируется после возврата из `cond_wait`. Если несколько потоков вызвали `cond_wait` для одной и той же переменной, то только один из них осуществляет повторную блокировку, поскольку остальные не смогут получить блокировку этой переменной. Поэтому, вызывая `cond_wait` только для синхронизации, нужно снять блокировку как можно скорее.

Функция `cond_signal` берет переменную, уже заблокированную в текущем потоке, и разблокирует один поток, ожидающий освобождения этой переменной по `cond_wait`. Если несколько потоков блокировано по этой переменной с помощью `cond_wait`, то только один из них будет разблокирован, и предсказать, какой именно, невозможно. Если нет потоков, блокированных `cond_wait` по этой переменной, событие удаляется.

Функция `cond_broadcast` действует подобно `cond_signal`, но разблокирует не один, а все потоки, блокированные `cond_wait` по этой переменной. (Разуме-

ется, остается в силе то, что одновременно переменная может быть заблокирована лишь в одном потоке.)

Функция cond_wait должна служить последним прибежищем для потока, если необходимое для него условие не выполняется. Функции cond_signal и cond_broadcast указывают на то, что условие изменяется. Предлагается следующий план действий: блокировка, затем проверка выполнения необходимого условия; если оно выполняется, то все хорошо, если нет – вызывается cond_wait и ожидается выполнение условия. Упор должен быть сделан на то, чтобы по возможности избежать блокировки. (Вообще это хороший совет для работы с потоками.)

Вот пример взаимной передачи управления между двумя потоками. Пусть вас не вводит в заблуждение то обстоятельство, что фактические условия находятся справа, в модификаторах команд: cond_wait вызываются, только если ожидаемое условие оказывается невыполненным.

```
use Thread qw(async cond_wait cond_signal);
my $wait_var = 0;
async {
    lock $wait_var;
    $wait_var = 1;
    cond_wait $wait_var  until $wait_var == 2;
    cond_signal($wait_var);
    $wait_var = 1;
    cond_wait $wait_var  until $wait_var == 2;
    $wait_var = 1;
    cond_signal($wait_var);
};

async {
    lock $wait_var;
    cond_wait $wait_var  until $wait_var == 1;
    $wait_var = 2;
    cond_signal($wait_var);
    cond_wait $wait_var  until $wait_var == 1;
    $wait_var = 2;
    cond_signal($wait_var);
    cond_wait $wait_var  until $wait_var == 1;
};
```

Другие модули для работы с потоками

Есть несколько модулей, работающих поверх примитивов cond_wait.

Очереди

Стандартный модуль Thread::Queue позволяет передавать объекты из одного потока в другой, не беспокоясь о блокировке или синхронизации. Его интерфейс значительно проще:

Метод	Использование
new	Создать новый объект Thread::Queue
enqueue	Поместить один или несколько скаляров в конец очереди
dequeue	Извлечь первый скаляр из начала очереди. Метод dequeue осуществляет блокировку, если очередь пуста

Обратите внимание на сходство очереди с обычным каналом. Различие лишь в том, что вместо байтов передаются целые скаляры, в том числе ссылки и благословленные (blessed) объекты!

Вот пример, извлеченный со страницы руководства *perlthrtut*:

```
use Thread qw/async/;
use Thread::Queue;

my $Q = Thread::Queue->new();
async {
    while (defined($datum = $Q->dequeue)) {
       print "Pulled $datum from queue\n";
    }
};

$Q->enqueue(12);
$Q->enqueue("A", "B", "C");
$Q->enqueue($thr);
sleep 3;
$Q->enqueue(\%ENV);
$Q->enqueue(undef);
```

Вот что будет выведено:

```
Pulled 12 from queue
Pulled A from queue
Pulled B from queue
Pulled C from queue
Pulled Thread=SCALAR(0x8117200) from queue
Pulled HASH(0x80dfd8c) from queue
```

Обратите внимание, что $Q находилась в области видимости, когда асинхронный поток был запущен через замыкание async. Потоки подчиняются общим для всех программ Perl правилам областей видимости. Приведенный пример не смог бы работать, если бы $Q была определена после вызова async.

Семафоры

Модуль Thread::Semaphore предоставляет объекты семафоров с защитой потоков и счетчиками, чтобы реализовать ваши любимые операции p() и v(). Поскольку большинство из нас не ассоциирует эти операции с голландскими словами *passeer* (*передавать*) и *verlaat* (*покидать*), в модуле они названы «down» и «up» соответственно. (В документации они иногда называются «wait» и «signal».) Поддерживаются следующие методы:

Метод	Использование
new	Создать новый объект Thread::Semaphore
down	Разместить один или несколько элементов
up	Удалить один или несколько элементов

Метод new создает новый семафор и устанавливает его счетчик в заданное значение. Если значение не указано, счетчику присваивается значение 1. (Это число представляет некоторый пул элементов, который может «опустеть», если все они размещены.)

```
use Thread::Semaphore;
$mutex = Thread::Semaphore->new($MAX);
```

Метод down уменьшает счетчик на заданное число или на 1, если число не указано. Его можно интерпретировать как попытку разместить некоторую часть или весь ресурс. Если счетчик семафора уменьшается до отрицательного числа, метод осуществляет блокирование, пока счетчик не станет больше или равен запрашиваемому числу. Вызов осуществляется так:

```
$mutex->down();
```

Метод up увеличивает счетчик на заданное число или на 1, если число не указано. Его можно интерпретировать как высвобождение некоторого количества ранее размещенного ресурса. При этом разблокируется по крайней мере один поток, заблокированный при попытке выполнить down для семафора, если только up увеличит счетчик семафора выше того значения, на которое down пытается его уменьшить. Вызывается метод так:

```
$mutex->up();
```

Другие стандартные модули для работы с потоками

Модуль Thread::Signal позволяет запустить поток, предназначенный для получения сигналов %SIG вашего процесса. При этом возникает все та же досадная проблема с ненадежностью сигналов в текущей реализации Perl и возможностью случайного получения дампа памяти при неосторожном их использовании.

Эти модули находятся в процессе разработки и могут не дать желаемых результатов при использовании в вашей системе. С другой стороны, могут и дать. Если не дадут результата, значит, кто-то вроде вас их еще не исправил. Вероятно, кто-то вроде вас должен вмешаться в дело и помочь.

18

Компиляция

Те, кто попал сюда в поисках компилятора Perl, могут удивиться, обнаружив, что он у них уже есть. Программа *perl* (обычно из каталога */usr/bin/perl*) уже содержит компилятор Perl. Возможно, кто-то думал, что это не так, и в этом случае им может быть приятно узнать, что мы предоставляем также *генераторы кода* (из лучших побуждений некоторые называют их «компиляторами»), обсуждаемые ближе к концу главы. Но сначала поговорим о том, как мы представляем себе Компилятор. В этой главе нам неизбежно придется вникать в некоторые технические подробности, причем кому-то они будут интересны, а кому-то – нет. Если вы принадлежите к последним, то можете считать, что вам предоставлена возможность попрактиковаться в быстром чтении.

Представьте себя дирижером, заказавшим партитуру большого оркестрового произведения. Получив свой заказ, вы обнаруживаете несколько десятков книжечек, по одной на каждого участника оркестра, в каждой из которых содержится партия только этого музыканта. С удивлением вы обнаруживаете, что главный экземпляр, содержащий все партии, отсутствует. Еще более удивительно, что те партии, которые есть, написаны на обычном английском языке, а не с помощью музыкальной нотации. Прежде чем составить программу выступления или даже раздать партии оркестрантам, необходимо перевести эти описания в прозе в обычную систему записи нот на линейках. Затем необходимо составить из отдельных частей одну гигантскую партитуру, чтобы получить представление о целом произведении.

Точно так же, если передать исходный код сценария Perl для исполнения программе *perl*, компьютер извлечет из этого не больше пользы, чем это сде-

лали бы музыканты из описания симфонии на английском языке. Прежде чем программа начнет выполняться, Perl должен скомпилировать[1] эти похожие на английский язык указания в специальное символическое представление. В результате этого программа еще не начнет выполняться, поскольку компилятор только компилирует. Подобно партитуре, лежащей перед дирижером, вашей программе даже после преобразования ее в инструкции, имеющие формат, который пригоден для интерпретации, требуется активный агент, чтобы интерпретировать эти инструкции.

Жизненный цикл программ на Perl

Жизненный цикл программы, написанной на Perl, можно разбить на четыре отдельные фазы, каждая из которых состоит из собственных этапов. Наибольший интерес представляют первая и последняя фазы, а две промежуточные обязательными не являются. Фазы изображены на рис. 18.1.

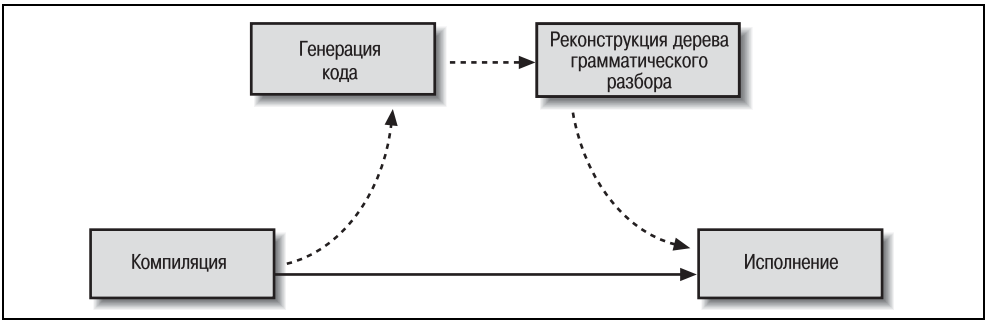


Рис. 18.1. Жизненный цикл программы Perl

1. *Фаза компиляции*

   Во время первой фазы, *фазы компиляции*, компилятор Perl преобразует программу в структуру данных, называемую *деревом грамматического разбора*. Наряду со стандартной технологией грамматического разбора Perl применяет другую, значительно более мощную: для управления дальнейшей компиляцией он использует блоки `BEGIN`. Они передаются интерпретатору для того, чтобы быть выполненными сразу после разбора, и он запускает их в порядке FIFO (первым пришел – первым ушел). В их число входят все объявления `use` и `no`, представляющие собой скрытые блоки `BEGIN`. Выполнение всех блоков `CHECK`, `INIT` и `END` откладывается компилятором на более позднее время.

   Лексические объявления отмечаются, но присвоения в них не производятся. На данном этапе компилируются все конструкции `eval` *`BLOCK`*, `s///e` и неинтерполируемые регулярные выражения и вычисляются константные выражения. Работа компилятора на этом заканчивается, если только его

[1] Или транслировать, трансформировать, преобразовать, перевоплотить, превратить.

не потребуется вызвать для работы позже. В конце этой фазы снова вызывается интерпретатор, чтобы выполнить запланированные блоки `CHECK` в порядке LIFO (обратном поступлению). Наличие или отсутствие блока `CHECK` определяет, будет совершен переход к фазе 2 или скачок к фазе 4.

2. *Фаза генерации кода (необязательная)*

   Блоки `CHECK` устанавливаются генераторами кода, поэтому данная необязательная фаза имеет место в случае явного использования одного из генераторов кода, описываемых далее в разделе «Генераторы кода» этой главы. Они преобразуют скомпилированную, но еще не запущенную программу в исходный код C или последовательность байт-кодов Perl – значений, выражающих внутренние инструкции Perl. Если вы решили создать исходный код C, то в конце концов из него можно создать файл на внутреннем языке машины[1], называемый *исполняемым образом* (*executable image*) или *загрузочным модулем* (*image file*).

   В этот момент ваша программа временно прекращает свои жизненные функции. Если вы создали загрузочный модуль, то можете перейти прямо к фазе 4; в противном случае потребуется подвергнуть полуфабрикат байт-кода восстановлению в фазе 3.

3. *Фаза реконструкции дерева грамматического разбора (необязательная)*

   Чтобы реанимировать программу, нужно воссоздать ее дерево грамматического разбора. Данная фаза наступает, только если производилась генерация кода и была выбрана генерация байт-кодов. Perl должен реконструировать из этой последовательности свои деревья грамматического разбора, прежде чем сможет выполнить программу. Perl не выполняет байт-коды непосредственно, это было бы слишком медленно.

4. *Фаза выполнения*

   Наконец, то, чего все ждали: прогон программы. Поэтому данная фаза имеет также название *фаза прогона* (*run phase*). Интерпретатор берет дерево грамматического разбора (полученное прямо от компилятора или косвенно через генерацию кода и последующую реконструкцию дерева грамматического разбора) и выполняет его. (Либо, если был создан исполняемый загрузочный модуль, он может выполняться как самостоятельная программа, поскольку в него встроен интерпретатор Perl.)

   В начале этой фазы, перед запуском основной программы, все запланированные блоки `INIT` выполняются в порядке FIFO. Затем выполняется основная программа. Интерпретатор может при необходимости снова вызвать компилятор, встретив команду `eval` *`STRING`*, `do` *`FILE`*, оператор `require`, конструкцию `s///ee` или поиск по шаблону с интерполяцией переменной, содержащей утверждение нулевой длины с кодом.

---

[1] Ваш исходный сценарий тоже является исполняемым файлом, но не на машинном языке, поэтому мы не называем его образом. Файл образа носит такое название потому, что это дословная копия машинных кодов, которые CPU умеет непосредственно выполнять.

По завершении выполнения основной программы будут наконец выполнены отложенные блоки `END`, на этот раз в порядке LIFO. Последним будет выполнен тот, который был обнаружен первым, и на этом все закончится. (Блоки `END` пропускаются, только если выполняется `exec` или процесс окажется прерванным неперехваченной критической ошибкой. Обычные исключительные ситуации не считаются критическими.)

Теперь обсудим эти фазы более подробно и в другом порядке.

Компилирование кода

Perl всегда находится в одном из двух режимов работы: либо компилирует вашу программу, либо выполняет ее, и никогда в обоих состояниях одновременно. На протяжении этой книги мы часто говорим, что некоторые события происходят во время компиляции, или что «компилятор Perl делает то-то и то-то». В других местах мы отмечаем, что нечто происходит во время выполнения, или «интерпретатор Perl делает то-то и то-то». Хотя можно ограничиться представлением и о компиляторе, и об интерпретаторе как просто о «Perl», осознание того, какую из двух ролей Perl играет в каждый данный момент, существенно для понимания причины происхождения тех или иных событий. Исполняемый модуль *perl* играет обе роли: сначала компилятора, а затем интерпретатора. (Есть и другие роли: *perl* является также оптимизатором и генератором кода. Иногда даже обманщиком, но шутит по-доброму.)

Важно также понять различие между фазой компиляции (compile phase) и временем компиляции (compile time), а также между фазой выполнения (run phase) и временем выполнения (run time). Типичная программа Perl проходит одну фазу компиляции, а затем одну фазу выполнения. «Фаза» – это широкое понятие, но время компиляции и время выполнения – понятия конкретные. В фазе компиляции производится, в основном, компилирование, но отчасти и выполнение – в блоках `BEGIN`. В фазе выполнения производится, в основном, выполнение, но могут осуществляться и действия времени компиляции, если встречаются операторы типа `eval` *`STRING`*.

В стандартной ситуации компилятор Perl прочитывает весь исходный текст программы, прежде чем начать выполнение. В это время Perl производит синтаксический анализ объявлений, команд и выражений, проверяя их синтаксическую допустимость.[1] Найдя синтаксическую ошибку, компилятор пытается как бы «не обращать на нее внимания», чтобы иметь возможность продолжить поиск ошибок в исходном тексте. Это удается не всегда: синтаксические ошибки имеют свойство порождать целый каскад ложных

[1] Формальных синтаксических диаграмм типа БНФ нет, но вы можете изучить файл *perly.y* в каталоге исходного текста Perl, содержащий грамматику *yacc* (1), используемую Perl. Советуем не трогать лексический анализатор, который известен тем, что вызывает нарушение аппетита у лабораторных крыс.

сообщений об ошибках. Встретив порядка 10 ошибок, Perl в отчаянии опускает руки.

В дополнение к интерпретатору, обрабатывающему блоки `BEGIN`, программу, при потворстве трех придирчивых агентов, обрабатывает компилятор. *Лексический анализатор* (*lexer*) высматривает в программе минимальные значащие единицы. В текстах, посвященных языкам программирования, их иногда называют «лексемами» или «маркерами» (lexemes, tokens). Лексический анализатор иногда называется tokener или scanner. Затем *синтаксический анализатор* (*parser*) пытается найти смысл групп этих лексем, собирая из них более крупные конструкции, такие как выражения и команды, исходя из грамматики языка Perl. *Оптимизатор* (*optimizer*) реорганизует эти более крупные образования и сокращает их в размерах, образуя более эффективные последовательности. Он тщательно выбирает, какую оптимизацию производить, не отвлекаясь на мелочи, поскольку компилятор Perl должен работать как молния, когда используется в режиме «загрузи и выполни».

Это происходит не на независимых один от другого этапах, а одновременно, при интенсивном общении агентов между собой. Лексическому анализатору иногда требуются советы синтаксического анализатора, чтобы определить, который из нескольких допустимых типов имеет рассматриваемая им лексема. (Как ни странно, лексическая область видимости (lexical scope) – одна из вещей, которые лексический анализатор *не* понимает, поскольку «лексический» имеет в данном случае другой смысл.) Оптимизатор тоже должен следить за действиями синтаксического анализатора, поскольку некоторые виды оптимизации нельзя осуществить, пока анализ не достигнет определенного момента, например, не обнаружил конец выражения, команды, блока или подпрограммы.

Может показаться странным, что компилятор Perl осуществляет все эти действия одновременно, а не одно за другим, но ведь это тот же самый запутанный процесс, который происходит при понимании естественного языка на лету – на слух или во время чтения. Ведь не все ждут конца главы, чтобы сообразить, каков смысл первого предложения. Можно представить себе такое соответствие:

Компьютерный язык	**Естественный язык**
Символ	Буква
Лексема	Морфема
Терм	Слово
Выражение	Фраза
Команда	Предложение
Блок	Параграф
Файл	Глава
Программа	Повесть

Если синтаксический анализ проходит успешно, компилятор рассматривает введенные данные как допустимую повесть – э-э, программу. Если при запуске программы указывается ключ -c, выводится сообщение «syntax OK» и осуществляется выход. В противном случае компилятор передает плоды своих трудов другим агентам. Эти «плоды» имеют форму *дерева грамматического разбора* (*parse tree*). Каждый «плод» на этом дереве, или *узел* (*node*), как его называют, представляет один из внутренних *кодов операции* (*opcodes*) Perl, а ветви дерева представляют историческую структуру его роста. В конце концов, узлы будут связаны между собой линейно, один за другим, чтобы обозначить порядок выполнения, в котором система этапа выполнения должна посетить эти узлы.

Каждый код операции является наименьшей единицей представления выполняемых команд в Perl. Если вы считаете, что выражение `$a = -($b + $c)` – это одна команда, то Perl считает, что здесь шесть разных кодов операции. В упрощенном виде дерево грамматического разбора для этого выражения выглядит так, как показано на рис. 18.2. Числа обозначают порядок, которому будет следовать система этапа выполнения Perl.

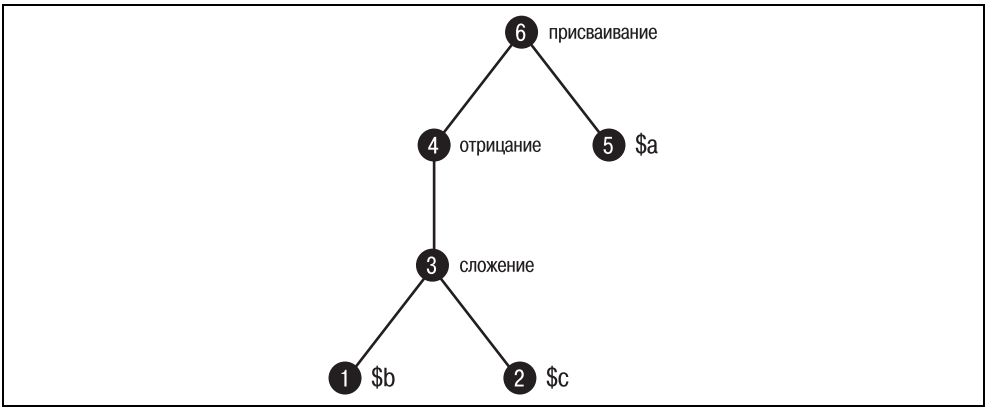


Рис. 18.2. Порядок выполнения кодов операций для выражения $a = –($b + $c)

Perl не является однопроходным компилятором, как это могло показаться. (Однопроходные компиляторы сильно облегчают жизнь компьютеру, но осложняют ее программисту.) В действительности это многопроходный оптимизирующий компилятор, осуществляющий по крайней мере три логически различных прохода, которые на практике чередуются. Проходы 1 и 2 осуществляются поочередно по мере того, как компилятор перемещается вверх-вниз по дереву синтаксического разбора во время его построения; проход 3 происходит, когда подпрограмма или файл полностью разобраны. Вот описание этих проходов:

1. *Проход 1: Восходящий синтаксический анализ*

   Во время этого прохода синтаксический анализатор *yacc*(1) строит дерево синтаксического разбора, используя лексемы, полученные от лексического анализатора (работу которого можно считать еще одним логическим

проходом). «Восходящий» означает, что в поле зрения анализатора сначала попадают листья дерева, а уже потом ветви и корень. Он действительно производит вычисления снизу вверх на рис. 18.2, поскольку мы изобразили корень вверху, в манере, характерной для ученых в области вычислительной техники (и лингвистов).

Для каждого создаваемого узла кода операции производится проверка правильности семантики, например, числа и типов аргументов при вызове встроенной функции. По мере того как вырисовываются части дерева, оптимизатор рассматривает возможность преобразования расположенных ниже поддеревьев. Например, узнав, что функции передается конкретное число аргументов, он может отбросить код операции, запоминающей число аргументов, переданных функции с переменным числом аргументов. Более важный вид оптимизации, известный как *свертывание констант* (*constant folding*), описан далее в этом разделе.

На этом проходе определяется порядок посещения узлов при последующем выполнении. Это ловкий фокус, поскольку первым посещаемым узлом почти никогда не является верхний. Компилятор создает временный цикл из кодов операций, при этом верхний узел указывает на первый посещаемый код операции. Когда код операции верхнего уровня включается в более крупную структуру, цикл кодов операций разрывается и образуется больший цикл с новым узлом в вершине. В конечном итоге цикл разрывается совсем, когда начальный код операции вставляется в некую другую структуру, например описатель подпрограммы. При вызове подпрограммы все же можно найти тот первый код операции, несмотря на то, что он находится далеко внизу дерева, как на рис. 18.2. Интерпретатору нет необходимости возвращаться вниз по дереву синтаксического разбора, чтобы определить начальную точку.

2. *Проход 2: Нисходящий оптимизатор*

   Если читать отрывок кода Perl (да и английского текста), то нельзя установить его контекст, не изучив окружающие лексические элементы. Иногда невозможно разобраться в происходящем без получения дополнительной информации. Не отчаивайтесь, однако, поскольку вы не одиноки: компилятору это тоже не удается. На этом проходе компилятор спускается вниз по только что построенному им поддереву, чтобы применить локальные оптимизации, самой примечательной из которых является *распространение контекста* (*context propagation*). Компилятор устанавливает для узлов, располагающихся ниже, надлежащий контекст (пустой, скаляр, список, ссылка или левое значение), определяемый текущим узлом. Лишние коды операций обнуляются, но не удаляются, поскольку изменять порядок выполнения уже слишком поздно. Мы предоставим третьему проходу возможность удалить их из временного порядка выполнения, определенного на первом проходе.

3. *Проход 3: Локальный оптимизатор*

   Некоторые участки кода имеют собственное пространство памяти, в котором хранятся переменные с лексическим контекстом. (На жаргоне Perl

такая область называется *scratchpad*, или временной памятью.) В число этих участков входят `eval STRING`, подпрограммы и целые файлы. С позиций оптимизатора более важно то, что все они имеют собственную точку входа. Это значит, что порядок выполнения известен нам в прямом направлении, но знать, что происходило раньше, мы не можем, поскольку конструкция могла быть вызвана из любого места. Поэтому, произведя синтаксический анализ такого участка, Perl запускает для работы с ним локальный оптимизатор (peephole optimizer). В отличие от предыдущих двух проходов, когда осуществлялось перемещение по дереву синтаксического разбора, в этом проходе код рассматривается в линейном порядке, поскольку сейчас, в сущности, предоставляется последняя возможность сделать это перед тем, как оторвать список кодов операций от синтаксического анализатора. Большинство оптимизирующих действий уже было произведено на первых двух проходах, но некоторые не могут быть выполнены на них.

Здесь происходят различные завершающие оптимизации, в том числе устанавливается окончательный порядок исполнения путем пропуска обнуленных кодов операций и определение мест, в которых расположенные рядом коды операций могут быть заменены чем-либо более простым. Важным видом оптимизации является распознавание сцепленных конкатенаций строк, поскольку хотелось бы избежать копирования строки всякий раз, когда к концу ее что-то добавляется. Этот проход не только оптимизирует, но и производит много «реальной» работы: отлавливает голые (bare) слова, генерирует предупреждения по поводу сомнительных конструкций, ищет код, который, вероятно, никогда не получит управления, разрешает ключи псевдохешей и отыскивает подпрограммы, вызываемые до компиляции своих прототипов.

4. *Проход 4: Генерация кода*

   Этот проход не является обязательным, обычно он не используется. Но если вызван один из трех генераторов кода — `B::Bytecode`, `B::C` или `B::CC`, дерево синтаксического разбора проходится еще один раз. Генераторы кода создают либо последовательность байт-кодов Perl, используемую впоследствии для реконструкции дерева синтаксического разбора, либо код на C, отражающий состояние дерева синтаксического разбора на этапе компиляции.

   Генерация кода C может быть двух различных видов. `B::C` просто воссоздает дерево синтаксического разбора и запускает его с помощью обычного цикла `runops()`, который сам Perl использует при выполнении. `B::CC` производит линеаризованный и оптимизированный C-эквивалент пути кода времени выполнения (напоминающий гигантскую таблицу переходов) и выполняет его.

Во время компиляции Perl оптимизирует код весьма многочисленными способами. Он перегруппировывает код, чтобы сделать его выполнение более эффективным. Он удаляет код, который никогда не будет выполнен, типа блока `if (0)` или `elsif` и `else` в блоке `if (1)`. Если используются переменные

лексического типа, объявленные в `my ClassName $var` или `our ClassName $var`, и пакет `ClassName` был установлен с директивой `use fields`, обращения к значениям констант из псевдохешей во время компиляции проверяются на наличие опечаток и преобразуются в обращения к массивам. Если вы задаете в операторе `sort` достаточно простую процедуру сравнения, такую как `{$a <=> $b}` или `{$b cmp $a}`, то она заменяется вызовом компилированного C-кода.

Из проводимых Perl оптимизаций наиболее впечатляет, возможно, то, как в нем при первой же возможности разрешаются константные выражения. Рассмотрим, например, дерево синтаксического разбора, показанное на рис. 18.2. Если бы узлы 1 и 2 оба были литералами или постоянными функциями, то узлы с 1 по 4 были бы заменены результатом этого вычисления; нечто похожее на этот результат изображено на рис. 18.3.

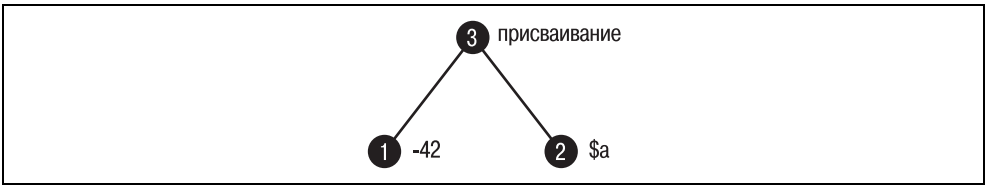


Рис. 18.3. Свертывание констант

Это называется *свертыванием констант* (*constant folding*). Свертывание констант не ограничивается такими простыми случаями, как, например, замена во время компиляции `2**10` на `1024`. При нем происходит также разрешение вызовов функций – как встроенных, так и объявляемых пользователями подпрограмм, удовлетворяющих критериям, описанным в разделе «Встраивание постоянных функций» главы 6 «Подпрограммы». Perl знает, какие из своих встроенных функций вызывать во время компиляции, что напоминает известную способность компиляторов FORTRAN вызывать свои внутренние функции. Поэтому если попытаться прологарифмировать `0.0` или извлечь корень из отрицательного числа, будет получена ошибка компиляции, а не времени исполнения, и интерпретатор вообще не будет запущен.[1]

Даже выражения с произвольной степенью сложности вычисляются на ранней стадии, что иногда влечет удаление целых блоков, как, например, следующего:

```
if (2 * sin(1)/cos(1) < 3 && somefn()) { whatever() }
```

Никакой код не генерируется для того, что никогда не будет вычисляться. Поскольку левая часть всегда принимает значение «ложь», ни `somefn`, ни `whatever` никогда не будут вызваны. (Поэтому не рассчитывайте, что вам удастся перейти на метки, находящиеся внутри блока: во время выполнения они просто не будут существовать.) Если бы `somefn` была встраиваемой посто-

[1] В действительности мы упростили изложение. Интерпретатор запущен будет, поскольку это необходимо для свертывания констант. Но произойдет это непосредственно на этапе компиляции, подобно тому как выполняются блоки `BEGIN`.

янной (inlinable constant) функцией, то даже изменение порядка вычислений на такой:

```
if (somefn() && 2 * sin(1)/cos(1) < 3)) { whatever() }
```

не изменило бы результата, поскольку все выражение по-прежнему решается во время компиляции. Если бы `whatever` была встраиваемой, ее вызов не производился бы ни во время выполнения, ни даже во время компиляции: ее значение было бы вставлено, как если бы являлось литеральной константой. Вы получили бы предупреждение о бесполезном применении константы в пустом контексте: «Useless use of a constant in void context». Оно может удивить, если не понять, что это константа. Однако если бы `whatever` была последней командой, выполняемой в функции, вызываемой в непустом контексте (согласно определению оптимизатора), то предупреждения вы бы не увидели.

Окончательный вид построенного дерева синтаксического разбора после всех оптимизаций можно увидеть с помощью команды *perl –Dx*. (Ключ `-D` требует специальной версии Perl с поддержкой отладки). См. ниже раздел, посвященный `B::Deparse`.

В общем, компилятор Perl старается (но не *слишком* сильно) оптимизировать код, с тем чтобы, когда придет время запуска программы, она выполнялась побыстрее. Пора бы уже и запустить вашу программу, чем мы сейчас и займемся.

Выполнение кода

В первом приближении Sparc-программы выполняются только на машинах Sparc, Intel-программы – только на машинах Intel, а Perl-программы – только на машинах Perl. Машина Perl обладает свойствами, которые Perl-программа считает идеальными для компьютера: автоматическое выделение и освобождение памяти, базовые типы данных в виде динамических строк, массивов и хешей без ограничений размера – или весьма к ним приближается. Задача интерпретатора Perl состоит в том, чтобы превратить тот компьютер, на котором он запущен, в одну из этих идеальных Perl-машин.

Эта фиктивная машина представляет собой иллюзию компьютера, специально разработанного исключительно для выполнения программ Perl. Каждый код операции, порожденный компилятором, является базовой командой в этом эмулированном наборе инструкций. Вместо аппаратного счетчика команд за очередностью выполнения кодов операций следит интерпретатор. Вместо указателя стека интерпретатор имеет собственный виртуальный стек. Это очень важный стек, поскольку виртуальная машина Perl (мы отказываемся называть ее PVM)[1] базируется на стеках. Коды операций Perl име-

[1] По-видимому, для того чтобы никто не думал, что речь идет о параллельной виртуальной машине, тоже PVM. – *Примеч. ред.* Или о Pneumonia Virus of Mice. – *Примеч. перев.*

ют внутреннее название *PP codes* («push-pop-коды»), поскольку для поиска операндов, обработки временных значений и хранения результатов они используют виртуальный стек интерпретатора.

Те, кому доводилось программировать на Forth или PostScript либо работать с научным калькулятором HP[1] с вводом в обратной польской (бесскобочной) нотации RPN (Reverse Polish Notation) вводом, знают, как работает машина со стековой организацией. В любом случае идея проста: чтобы сложить 3 и 4, действия производятся в порядке `3 4 +` вместо более привычного `3 + 4`. На языке стека это означает, что сначала в стек проталкивается `3`, затем `4`, а потом `+` выталкивает со стека оба аргумента, складывает их и проталкивает `7` обратно в стек, где результат будет находиться, пока вы не сделаете с ним что-нибудь еще.

В сравнении с компилятором Perl интерпретатор Perl является простой и почти скучной программой. Он всего лишь перебирает поочередно скомпилированные коды операций и отправляет их среде выполнения Perl, т. е. виртуальной машине Perl. Всего лишь небольшая C-программа, не так ли?

На самом деле она совсем не скучная. Виртуальная машина Perl следит вместо вас за большим объемом динамического контекста. Perl поддерживает много стеков, разбираться в которых не обязательно, но мы все же перечислим их здесь, чтобы произвести на читателя впечатление:

стек операндов

Об этом стеке мы уже говорили.

стек сохранения

Здесь сохраняются локализованные значения в ожидании восстановления. Многие внутренние процедуры локализуют значения невидимым для вас образом.

стек области видимости

Облегченный динамический контекст, определяющий, когда должны выталкиваться данные из стека сохранения.

стек контекста

Полновесный динамический контекст: последовательность вызовов, приводящая туда, где вы сейчас находитесь. По этому стеку проходит функция `caller`. Функции управления циклами просматривают этот стек в поисках цикла, которым нужно управлять. При перемещении назад по стеку контекста происходит соответствующее перемещение по стеку видимости, в результате чего восстанавливаются локальные переменные со стека сохранения, даже если выход из предыдущего контекста был выполнен каким-нибудь нехорошим способом вроде возбуждения исключительной ситуации и *longjmp*(3).

---

[1] Аналоги выпускались, вроде бы, еще в СССР: модели БЗ-34, МК 61, МК 52. – *Примеч. науч. ред.*

стек jumpenv

Стек контекстов *longjmp*(3), позволяющий возбуждать исключительные ситуации и выходить поспешно.

стек возврата

Точка, из которой мы вошли в эту подпрограмму.

стек маркеров (mark stack)

Место, где начинается текущий список разных аргументов на стеке операндов.

стеки временной памяти лексических переменных для рекурсии

Место хранения лексических переменных и других элементов «временного учета» при рекурсивном вызове подпрограмм.

И конечно, имеется стек C, в котором хранятся все переменные C. Perl на практике старается не полагаться на стек C при хранении запоминаемых величин, поскольку при *longjmp*(3) не происходит правильного восстановления таких величин.

Все это сказано к тому, что обычная точка зрения на интерпретатор – программу, интерпретирующую другую программу, – оказывается, к сожалению, неадекватной для описания всего здесь происходящего. Да, есть некоторый C-код, реализующий некоторые коды операций, но, произнося слово «интерпретатор», мы подразумеваем нечто большее – так же, как, произнося «музыкант», мы подразумеваем нечто большее, чем набор инструкций ДНК для превращения нот в звуки. Музыканты – реальные живые организмы со своим «состоянием». То же относится к интерпретаторам.

Конкретно, весь этот динамический и лексический контекст, глобальные таблицы символов, деревья синтаксического разбора плюс поток выполнения и есть то, что мы называем интерпретатором. Как контекст выполнения интерпретатор начинает существовать еще раньше, чем компилятор, и может существовать в рудиментарном виде даже в процессе создания компилятором контекста интерпретатора. На практике именно это и происходит, когда компилятор вызывает интерпретатор для выполнения блоков `BEGIN` и подобных. А интерпретатор может воспользоваться компилятором, чтобы достраивать себя еще дальше. Всякий раз, когда определятся новая подпрограмма или загружается новый модуль, некоторая виртуальная машина Perl, называемая интерпретатором, переопределяет себя заново. Невозможно точно сказать, которая из двух программ – компилятор или транслятор – работает, потому что они взаимодействуют при управлении процессом самонастройки, который мы обычно называем «запуском сценария Perl». Это как самонастройка мозга ребенка. Кто ее осуществляет – ДНК или нейроны? Мы думаем – и та и другие, а также некоторые данные, вводимые посторонними программистами.

В одном процессе можно запустить несколько интерпретаторов. При этом они могут совместно использовать деревья синтаксического разбора, если запускаются путем клонирования существующего интерпретатора, а не соз-

дания нового интерпретатора с нуля. Можно также в одном интерпретаторе запускать несколько потоков, которые будут иметь общий доступ не только к деревьям синтаксического разбора, но и к глобальным символам – см. главу 17 «Потоки».

Однако большинство программ Perl использует только один интерпретатор Perl для выполнения скомпилированного кода. И хотя можно запустить несколько независимых интерпретаторов Perl в одном процессе, в настоящее время API для этого доступен только из C.[1] Каждый отдельный интерпретатор Perl играет роль отдельного процесса, но создание его обходится дешевле, чем для совершенно нового процесса. Вот почему модуль расширения Apache `mod_perl` достигает такой высокой производительности: при запуске сценария CGI под `mod_perl` этот сценарий оказывается уже скомпилированным в коды операций Perl и не требует перекомпиляции, но что еще более важно, нет необходимости запускать новый процесс, а именно это и есть узкое место. Apache инициализирует новый интерпретатор Perl в существующем процессе и передает ему на выполнение ранее скомпилированный код. Конечно, как всегда, в действительности все сложнее. Подробнее о `mod_perl` можно прочесть в книге «Writing Apache Modules with Perl and C» (Создание модулей Apache на Perl и C), O'Reilly, 1999.

Во многие другие приложения, например *nvi*, *vim* и *innd*, можно встраивать интерпретаторы Perl – все перечислить здесь не удастся. Есть ряд коммерческих продуктов, в которые встроено ядро Perl, но это не рекламируется. Они просто используют Perl внутри себя, поскольку он позволяет им красиво решать их задачи.

Серверы компиляторов

Что ж, если Apache может скомпилировать программу Perl и выполнить ее позже, почему вы не можете поступить так же? Apache и другие программы со встроенными интерпретаторами Perl делают это просто: они не сохраняют дерево синтаксического разбора во внешнем файле. Если такой подход вас устраивает и вы не возражаете против использования C API с этой целью, то можете осуществить то же самое. См. раздел «Встраивание Perl» главы 21 «Изнутри и снаружи», чтобы узнать, как получить доступ к Perl из внешней среды C.

Если вы не хотите идти таким путем или вам нужно что-то еще, то есть несколько других вариантов. Вместо того чтобы передавать коды операций, созданные компилятором Perl, непосредственно на интерпретатор Perl, можно активизировать некоторые другие серверы (backends). Эти серверы могут сериализовать и записать скомпилированные коды операций во внешние файлы или даже преобразовывать их в различного типа C-код.

---

[1] За одним исключением: версия Perl 5.6.0 может осуществлять клонирование интерпретаторов при поддержке эмуляции ветвления на Microsoft Windows. Вполне может оказаться, что, когда вы это читаете, уже появился Perl API к «ithreads», как они называются.

Имейте в виду, что эти генераторы кода являются в значительной степени экспериментальными средствами и в промышленной среде разработки полагаться на них не следует. На самом деле полагаться на них не следует и в непромышленной среде, и работать они будут от случая к случаю. Теперь, когда мы настроили вас так, что любой успех превзойдет всякие ваши ожидания, можно спокойно рассказать, как работают эти серверы.

Некоторые из backend-модулей являются генераторами кода, например `B::Bytecode`, `B::C` и `B::CC`. Другие, в сущности, являются средствами анализа и отладки, например `B::Deparse`, `B::Lint` и `B::Xref`. Помимо серверов (backends) в стандартную версию входят несколько других низкоуровневых модулей, которые могут представлять интерес для будущих авторов средств разработки программ на Perl. Другие backend-модули можно найти в CPAN, в том числе (на момент написания данной книги) `B::Fathom`, `B::Graph`, `B::JVM::Jasmin` и `B::Size`.

Когда компилятор Perl используется не на входе интерпретатора, а в других целях, между ним и различными backend-модулями размещается модуль `O` (с помощью файла *O.pm*). Backend не вызывается прямо, вместо него вызывается среднее звено, которое, в свою очередь, вызывает указанный сервер (backend). Если имеется, скажем, модуль `B::Backend`, то вызвать его с нужным сценарием можно так:

```
% perl -MO=Backend SCRIPTNAME
```

Некоторые backends принимают параметры, которые указываются так:

```
% perl -MO=Backend, OPTS SCRIPTNAME
```

У некоторых серверов (backends) уже есть интерфейсы, позволяющие вызывать среднее звено, так что вам нет необходимости запоминать их MO. В частности, *perlcc*(1) вызывает этот генератор кода, который может оказаться довольно громоздким для запуска.

Генераторы кода

Со всей определенностью нужно сказать, что все три имеющиеся в настоящее время сервера (backends), преобразующие коды операций Perl в некоторый другой формат, являются экспериментальными. (Мы уже говорили об этом ранее, но считаем необходимым напомнить еще раз.) Даже когда им случается сгенерировать код, который выполняется правильно, результирующие программы могут потребовать больше дискового пространства, памяти и ресурсов CPU, чем в обычном случае. Это область продолжающихся исследований и разработок. Положение будет со временем улучшаться.

Генератор Bytecode

Модуль `B::Bytecode` выводит коды операций дерева синтаксического разбора в независимой от платформы кодировке. Можно взять сценарий Perl, ском-

пилированный в байт-коды, и перенести его на любую машину, на которой установлен Perl.

Стандартная, но пока экспериментальная команда *perlcc*(1) позволяет преобразовать исходный код на Perl в скомпилированную в байты программу Perl. Нужно лишь выполнить команду:

```
% perlcc -b -o pbyscript srcscript
```

После этого можно непосредственно «выполнять» полученный *pbyscript*. Начало этого файла выглядит примерно так:

```
#!/usr/bin/perl
use ByteLoader 0.03;
^C^@^E^A^C^@^@^@^A^F^@^C^@^@^@^B^F^@^C^@^@^@^C^F^@^C^@^@^@
B^@^@^@^H9^A8M-^?M-^?M-^?M-^?7M-^?M-^?M-^?M-^?6^@^@^@^A6^@
^G^D^D^@^@^@^KR^@^@^@^HS^@^@^@^HV^@M-2W<^FU^@^@^@^@X^Y@Z^@
...
```

Виден небольшой заголовок сценария, за которым следуют двоичные данные. Это может выглядеть тайной магией, но программа, преобразующая эти данные (их «dwimmer»), по большей части не сложна. Модуль использует технологию *входного фильтра* (*source filter*), чтобы преобразовать исходный текст, перед тем как представить его Perl. Входной фильтр – своего рода препроцессор, применяемый ко всему нижеследующему содержимому текущего файла. Он выполняет не только простые преобразования, осуществляемые, например, макропроцессорами *cpp*(1) и *m4*(1): никаких ограничений не накладывается. Есть входные фильтры для расширения синтаксиса Perl, сжатия и шифрования исходного кода и даже записи программ Perl на латинском языке. E perlibus unicode; cogito, ergo substr; carp dbm, et al. Caveat scriptor – пусть автор сценария будет бдителен.

Модуль `ByteLoader` – это входной фильтр, способный дизассемблировать переведенные в последовательную форму коды операций, произведенные `B::Bytecode`, и восстанавливать исходное дерево синтаксического разбора. Воссозданный код Perl образует текущее дерево синтаксического разбора без помощи компилятора. Когда интерпретатор находит эти коды операций, он просто выполняет их, как если бы они только того и ждали.

Генераторы кода на C

Оставшиеся генераторы кода, `B::C` и `B::CC`, создают не преобразованные в последовательную форму коды операций Perl, а код C. Генерируемый ими код не отличается хорошей читаемостью, и вы ослепнете, если попытаетесь в нем разобраться. Использовать их для того, чтобы вставлять в большую программу на C маленькие фрагменты, оттранслированные из Perl в C, не удастся. Подробнее об этом см. главу 21.

Модуль `B::C` просто выводит структуры данных C, необходимые для воссоздания целиком среды времени выполнения Perl. В результате получается специализированный интерпретатор со всеми предварительно инициализи-

рованными структурами данных, построенными компилятором. В некоторых отношениях получаемый код похож на то, что создает `B::Bytecode`. Оба они выполняют простое транслирование деревьев кодов операций, создаваемых компилятором, но если `B::Bytecode` делает это в символической форме, чтобы позже воссоздать и передать работающему интерпретатору Perl, то `B::C` записывает эти коды операций на C. После компиляции этого кода компилятором C и компоновки с библиотекой Perl для выполнения результирующей программы не требуется установка интерпретатора Perl на машине. (Могут, однако, потребоваться некоторые библиотеки совместного доступа, если сборка не была полностью статической.) Эта программа, в сущности, не отличается от обычного интерпретатора Perl, выполняющего сценарий. Она просто предварительно скомпилирована как самостоятельный исполняемый модуль.

Модуль `B::CC` пытается пойти значительно дальше. Начало генерируемого им файла весьма похоже на то, что создает `B::C`,[1] но затем всякое сходство прекращается. В коде, создаваемом `B::C`, имеется большая таблица кодов операций на C, которая обрабатывается так, как это делал бы сам интерпретатор, в то время как C-код, генерируемый `B::CC`, записывается в порядке, соответствующем логике выполнения вашей программы. В нем есть даже C-функции, соответствующие всем функциям вашей программы. Производится некоторая оптимизация по типам переменных. Некоторые контрольные задачи выполняются вдвое быстрее, чем при обычной интерпретации. Это наиболее претенциозный из имеющихся в настоящее время генераторов кода и самый многообещающий. Не случайно, что из всех трех он наименее устойчив в работе.

Студенты, изучающие вычислительную технику и не выбравшие еще тему дипломного проекта, могут не искать ничего иного. Здесь масса необработанных алмазов, ждущих огранки.

Средства разработки кода

У модуля `O` есть много интересных возможностей помимо предоставления данных для невыносимо недоработанных генераторов кода. Предоставляя относительно легкий доступ к результатам работы компилятора Perl, этот модуль облегчает создание других средств, которым нужно знать все о программе Perl.

Модуль `B::Lint` получил свое название от *lint*(1), верификатора программ C. Он ищет в программах сомнительные конструкции, о которые часто спотыкаются новички, но которые обычно не влекут вывод предупреждений. Модуль вызывается непосредственно:

```
% perl -MO=Lint,all myprog
```

[1] Как и все прочее, если вы потеряли зрение. Но мы же предупреждали, чтобы вы не подглядывали!

В настоящее время осуществляется всего несколько видов проверок, например, на использование массивов в неявных скалярных контекстах, использование переменных по умолчанию и обращение к идентификаторам в других пакетах (номинально закрытым), начинающимся с символа _. Подробности см. в *B::Lint*(3).

Модуль `B::Xref` генерирует списки перекрестных ссылок для объявления и использования всех переменных (как с глобальной, так и с лексической областями видимости), подпрограмм и форматов с разбивкой по файлам и подпрограммам. Модуль вызывается так:

```
% perl -MO=Xref myprog > myprof.pxref
```

Вот пример части создаваемого отчета:

```
Subroutine parse_argv
  Package (lexical)
    $on                i113, 114
    $opt               i113, 114
    %getopt_cfg        i107, 113
    @cfg_args          i112, 114, 116, 116
  Package Getopt::Long
    $ignorecase        101
    &GetOptions        &124
  Package main
    $Options           123, 124, 141, 150, 165, 169
    %$Options          141, 150, 165, 169
    &check_read        &167
    @ARGV              121, 157, 157, 162, 166, 166
```

Здесь видно, что подпрограмма `parse_argv` имеет четыре собственные лексические переменные; кроме того, она обращается к глобальным идентификаторам из пакета `main` и из `Getopt::Long`. Числа указывают на строки, в которых используется объект: префикс `i` означает, что первое появление объекта имело место в строке с номером, который за ним следует, а префикс `&` указывает на вызов подпрограммы в соответствующей строке. Разыменования перечисляются отдельно, поэтому показаны как `$Options`, так и `%$Options`.

Модуль `B::Deparse` – изящная программа печати, позволяющая снять пелену таинственности с кода Perl и понять, какие преобразования произвел с кодом оптимизатор. Ниже показано, например, какие значения по умолчанию использует Perl для различных конструкций:

```
% perl -MO=Deparse -ne 'for (1 .. 10) { print if -t }'
LINE: while (defined($_ = <ARGV>)) {
    foreach $_ (1 .. 10) {
        print $_ if -t STDIN;
    }
}
```

Ключ `-p` расставляет скобки, что позволяет видеть, какие приоритеты операторов использует Perl:

```
% perl -MO=Deparse,-p -e 'print $a ** 3 + sqrt(2) / 10 ** -2 ** $c'
print(((($a ** 3) + (1.4142135623731 / (10 ** (-(2 ** $c))))));
```

С помощью ключа -q можно увидеть, в какие примитивы компилируются интерполируемые строки:

```
% perl -MO=Deparse,-q -e '"A $name and some @ARGV\n"'
'A ' . $name . ' and some ' . join($", @ARGV) . "\n";
```

А ниже показано, как Perl в действительности компилирует состоящий из трех частей цикл `for` в цикл `while`:

```
% perl -MO=Deparse -e 'for ($i=0;$i<10;$i++) { $x++ }'
$i = 0;
while ($i < 10) {
    ++$x;
}
continue {
    ++$i
}
```

Можно даже применить `B::Deparse` к файлу байт-кодов Perl, сгенерированному командой *perlcc -b*, и заставить его декомпилировать исполняемый модуль. Переведенные в последовательную форму коды операций Perl читать, может быть, трудновато, но стойкого шифрования в них нет.

Компилятор и интерпретатор: авангардизм и ретро

Думать обо всем нужно в должное время, иногда до того, иногда после. Этот момент может наступить и в середине. Perl не притворяется, что знает, когда наступает подходящий момент для размышлений, поэтому он предоставляет программисту ряд возможностей известить себя об этом. Иногда Perl'у становится ясно, что нужно немного подумать, но не ясно, о чем именно, поэтому у него должен быть способ спросить об этом вашу программу, которая отвечает на такие вопросы путем определения подпрограмм с именами, соответствующими тому, что Perl пытается выяснить.

Не только компилятор может вызвать интерпретатор, когда хочет быть предусмотрительным, но и интерпретатор может снова вызвать компилятор, когда ему нужно пересмотреть историю. Есть несколько операторов, которые можно использовать в программе для обратного вызова компилятора. Как и компилятор, интерпретатор может вызывать подпрограммы по именам, когда ему нужно выяснить положение вещей. Весь этот обмен, происходящий между компилятором, интерпретатором и программой, требует, чтобы вы знали, когда и какие события происходят. Сначала мы обсудим вопрос о том, когда запускаются эти именованные подпрограммы.

В главе 10 «Пакеты» мы видели, как запускается процедура AUTOLOAD при обращении к функции, которая не определена в этом пакете. В главе 12 «Объекты» мы встретились с методом DESTROY, вызываемым, когда Perl намеревается автоматически возвратить память объекта. И в главе 14 «Связанные переменные» мы столкнулись со многими функциями, неявно вызываемыми при обращении к связанной переменной.

Все эти подпрограммы используют соглашение записывать заглавными буквами имя подпрограммы, которая автоматически запускается компилятором или интерпретатором. С различными этапами жизненного цикла программы связаны еще четыре такие подпрограммы: BEGIN, CHECK, INIT и END. Ключевое слово sub перед их объявлениями не обязательно. Возможно, их правильнее было бы называть «блоками», поскольку в ряде отношений они ведут себя скорее как именованные блоки, а не настоящие подпрограммы.

Например, в отличие от обычных подпрограмм, эти блоки допускается объявлять многократно, поскольку Perl следит за тем, когда их вызывать, и вам не нужно вызывать их по имени. (Отличие от обычных подпрограмм состоит также в том, что shift и pop действуют так, как если бы вы находились в главной программе, поэтому по умолчанию они воздействуют на @ARGV, а не на @_.)

Эти четыре блока идут в следующем порядке:

BEGIN

Запускается ASAP (as soon as parsed – сразу после синтаксического разбора), как только встретится во время компиляции и прежде чем компилировать оставшуюся часть файла.

CHECK

Запускается по завершении компиляции, но перед запуском программы. (CHECK может означать «контрольную точку», «двойную проверку» или даже «останов».)

INIT

Запускается в начале выполнения перед началом основной логики программы.

END

Запускается в конце выполнения сразу после завершения программы.

Если объявить несколько таких подпрограмм с одинаковыми именами, даже в разных модулях, то все BEGIN выполняются перед всеми CHECK, которые все выполняются раньше всех INIT, которые все выполняются раньше всех END, которые выполняются все в самую последнюю очередь после завершения основной программы. Если блоков BEGIN и INIT несколько, они выполняются в порядке объявления (FIFO), а CHECK и END выполняются в порядке, обратном объявлению (LIFO).

Проще всего, вероятно, посмотреть это на следующем примере:

```
#!/usr/bin/perl -l

print       "начало выполнения main";
die         "завершение main\n";
die         "XXX: не достигнут\n";
END         { print "1-й END: конец выполнения"   }
CHECK       { print "1-й CHECK: конец компиляции" }
INIT        { print "1-й INIT: начало выполнения"  }
END         { print "2-й END: конец выполнения"   }
BEGIN       { print "1-й BEGIN: продолжение компиляции" }
INIT        { print "2-й INIT: начало выполнения"  }
BEGIN       { print "2-й BEGIN: продолжение компиляции" }
CHECK       { print "2-й CHECK: конец компиляции" }
END         { print "3-й END: конец выполнения"   }
```

Эта демонстрационная программа на выходе дает следующее:

```
1-й BEGIN: продолжение компиляции
2-й BEGIN: продолжение компиляции
2-й CHECK: конец компиляции
1-й CHECK: конец компиляции
1-й INIT: начало выполнения
2-й INIT: начало выполнения
начало выполнения main
завершение main
3-й END: конец выполнения
2-й END: конец выполнения
1-й END: конец выполнения
```

Поскольку блок BEGIN выполняется немедленно, он может ввести объявления подпрограмм, определения и осуществить импорт еще до того, как будет скомпилирована остальная часть файла. В результате может измениться то, как компилятор осуществляет синтаксический анализ оставшейся части файла, особенно если импортируются определения подпрограмм. По крайней мере, объявление подпрограммы позволяет использовать ее в качестве спискового оператора, делая необязательным применение скобок. Если импортированная подпрограмма объявлена с прототипом, обращения к ней могут быть проанализированы как встроенные и даже переопределять встроенные подпрограммы с тем же именем, сообщая им новую семантику. К объявлению use можно относиться просто как к блоку BEGIN.

Напротив, блоки END выполняются как можно *позже*: когда программа покидает интерпретатор Perl, даже если это результат неперехваченного die или другой фатальной исключительной ситуации. Есть два случая, в которых блок END (или метод DESTROY) пропускается. Он не выполняется, если текущий процесс вместо выхода переходит из одной программы в другую посредством exec. Процесс, «выброшенный на берег» из-за неперехваченного сигнала, тоже пропускает свои процедуры END. (См. описание директивы use sigtrap в главе 31 «Модули прагм», там рассматривается преобразование перехваченных сигналов в исключительные ситуации. Общие сведения об об-

работке сигналов приведены в разделе «Сигналы» главы 16 «Связь между процессами».) Можно предотвратить выполнение всех END, вызвав POSIX::_exit, скажем, kill -9, $$, или просто выполнив через exec какую-нибудь безобидную программу вроде */bin/true* в системах Unix.

Внутри блока END переменная $? содержит статус, с которым собирается выйти программа. Можно изменить значение $? внутри блока END, чтобы программа завершилась с другим значением. Не измените случайно $?, запустив другую программу через system или обратный штрих.

Если в файле несколько блоков END, то они выполняются в порядке, *обратном* их определению. Таким образом, последний определенный блок END будет выполнен первым по завершении программы. Это позволяет организовать правильное вложение попарно связанных друг с другом блоков BEGIN и END. Например, если основная программа и загружаемый ею модуль имеют свои парные подпрограммы BEGIN и END, как показано ниже:

```
BEGIN { print "main begun" }
END { print "main ended" }
use Module;
```

и такие объявления в модуле:

```
BEGIN { print "module begun" }
END { print "module ended" }
```

то основная программа знает, что ее BEGIN всегда будет выполнен первым, а ее END — последним. (Да, BEGIN является, в действительности, блоком этапа компиляции, но аналогичные принципы применимы к парным блокам INIT и END на этапе выполнения.) Этот принцип рекурсивно выполняется для любого файла, содержащего другой, если в обоих есть такие объявления. Данное свойство вложенности позволяет использовать блоки как конструкторы и деструкторы пакетов. В каждом модуле могут быть свои функции установки и демонтажа, которые Perl вызовет автоматически. Благодаря этому программисту не нужно помнить, применяя конкретные библиотеки, какой код инициализации или уборки мусора должен вызываться и когда. Объявления модуля обеспечивают эти события.

Если рассматривать eval *STRING* как *обратный* вызов интерпретатором компилятора, то BEGIN можно рассматривать как *прямой* вызов из компилятора в интерпретатор. В том и в другом случаях временно откладывается текущая работа и переключается режим работы. Когда мы говорим, что блок BEGIN выполняется как можно раньше, мы имеем в виду, что он выполняется, как только будет полностью определен, даже раньше, чем будет проанализирована оставшаяся часть файла. Поэтому блоки BEGIN всегда выполняются на этапе компиляции, а не выполнения. После прогона блока BEGIN он сразу становится неопределенным, а использовавшийся в нем код возвращается в пул памяти Perl. При всем желании вызвать блок BEGIN как подпрограмму не удастся, потому что к тому времени, когда вы сможете это сделать, его уже не будет в живых.

Аналогично блокам BEGIN блоки INIT запускаются перед началом этапа выполнения Perl в порядке «первым прибыл, первым обслужен» (FIFO). Например, генераторы кода, документированные в *perlcc*, используют блоки INIT для инициализации и разрешения указателей на XSUB. Боки INIT очень похожи на блоки BEGIN, но позволяют программисту отделить конструктор, выполняемый на этапе компиляции, от конструктора, выполняемого на этапе исполнения. При непосредственном запуске сценария это не очень важно, поскольку в любом случае каждый раз вызывается компилятор. Но если компиляция отделена от исполнения, это различие может быть критическим. Компилятор может быть вызван один раз, а полученный исполняемый модуль может вызываться многократно.

Аналогично блокам END блоки CHECK запускаются сразу по окончании этапа компиляции Perl, но до начала этапа исполнения, в порядке, обратном поступлению (LIFO). Блоки CHECK полезны для того, чтобы «свернуть» компилятор, подобно тому как блоки END удобно применять, для того чтобы «свернуть» программу. В частности, серверы используют блоки CHECK как перехватчики для запуска своих генераторов кода. Все, что им нужно сделать – это вставить блок CHECK в собственный модуль, и он будет запущен в нужный момент, а вам не потребуется вставлять CHECK в свою программу. Поэтому программисту редко требуется самому писать блок CHECK, если только он не создает свой модуль такого типа.

Объединяя все сказанное, табл. 18.1 перечисляет различные конструкции с подробностями о том, когда они компилируются и когда выполняют код, обозначенный «...».

Таблица 18.1. Что когда происходит

Блок или выражение	Компилируется в фазе	Перехватывает ошибки компиляции	Выполняется в фазе	Перехватывает ошибки исполнения	Политика запуска вызова
use ...	C	Нет	C	Нет	Сейчас
no ...	C	Нет	C	Нет	Сейчас
BEGIN {...}	C	Нет	C	Нет	Сейчас
CHECK {...}	C	Нет	C	Нет	Поздний
INIT {...}	C	Нет	R	Нет	Ранний
END {...}	C	Нет	R	Нет	Поздний
eval {...}	C	Нет	R	Да	Inline
eval "..."	R	Да	R	Да	Inline
foo(...)	C	Нет	R	Нет	Inline
sub foo {...}	C	Нет	R	Нет	Вызов в любое время
eval "sub {...}"	R	Да	R	Нет	Поздний вызов
s/pat/.../e	C	Нет	R	Нет	Inline
s/pat/"..."/ee	R	Да	R	Да	Inline

Теперь, уяснив партитуру, вы сможете сочинять и исполнять свои пьесы для Perl с большей уверенностью.

19

Интерфейс командной строки

Эта глава посвящена тому, как осуществить наводку Perl в нужном направлении, прежде чем открывать пальбу. Есть несколько способов наводки Perl, главными двумя из которых являются использование ключей в командной строке и переменных окружения. Ключи предоставляют более непосредственный и точный способ нацеливания конкретной команды. Переменные окружения чаще используются для настройки общей политики.

Обработка команд

Счастье, что Perl эволюционировал в мире Unix, поскольку благодаря этому синтаксис его вызова работает достаточно хорошо в интерпретаторах команд и других операционных систем. Большинство интерпретаторов команд в состоянии работать со списком слов, представляющих собой аргументы, и не смущается, если первый символ аргумента начинается с символа «-». Есть, правда, неприятные моменты, в которых можно запутаться при переходе от одной системы к другой. Например, нельзя использовать одиночные кавычки в MS-DOS, как это делается в Unix. А в таких системах, как VMS, некоторым программам-упаковщикам придется нелегко при эмулировании переадресации ввода-вывода Unix. Групповые символы интерпретируются как групповые символы. Однако после того как вы разберетесь в этих различиях, Perl станет работать с ключами и аргументами сходным образом в любой операционной системе.

Даже если у вас отсутствует собственно интерпретатор команд, легко запускать программы Perl из любых других программ, написанных на любом языке. Вызывающая программа может передавать аргументы не только обычным способом, но и через переменные окружения и, если позволяет операционная система, через унаследованные дескрипторы файлов (см. раздел «Передача дескрипторов файлов» главы 16 «Связь между процессами». Даже экзотические механизмы передачи аргументов легко реализовать в отдельном модуле, подключив его к программе Perl с помощью обычной директивы `use`.

Perl анализирует аргументы командной строки обычным образом.[1] Это значит, что он рассчитывает получить ключи (слова, начинающиеся со знака «-») в начале командной строки. Затем обычно следуют название сценария и дополнительные аргументы, которые должны быть переданы сценарию. Некоторые из этих дополнительных аргументов сами могут выглядеть как ключи, но их должен обрабатывать сценарий, поскольку Perl прекращает разбор ключей, встретив аргумент, не являющийся ключом, или специальный ключ «--», означающий, что этот ключ – последний.

Perl предоставляет некоторую свободу в размещении исходного кода программы. В случае маленьких одноразовых заданий можно целиком поместить программу Perl в командной строке. Более объемные и «долгоживущие» задачи можно оформлять как сценарии Perl в отдельных файлах. Сценарий для компиляции и запуска Perl может быть указан одним из следующих трех способов:

1. Построчно в командной строке с помощью ключей `-e`, например:

```
% perl -e "print 'Hello, World.'"
Hello, World.
```

2. Включением в файл – задается первым параметром в командной строке. Системы, поддерживающие обозначение `#!` в первой строке файла с исполняемым сценарием, вызывают интерпретатор, указанный за данным обозначением.
3. Передан косвенно через стандартный ввод. Этот метод срабатывает только при отсутствии имен файлов среди аргументов; для передачи аргументов сценарию, передаваемому через стандартный ввод, нужно использовать метод 2, явно указывая в качестве имени сценария «-». Например:

```
% echo "print qq(Hello, @ARGV.)" | perl - World
Hello, World.
```

При использовании методов 2 и 3 Perl начинает синтаксический анализ входного файла с самого начала, если только не задан ключ `-x`, когда он ищет первую строку, начинающуюся с `#!` и содержащую слово «perl», и начинает анализ с этого места. Это полезно при запуске сценария, встроенного

[1] Исходя из предположения, что вы также считаете Unix и шаблонной, и стильной системой.

в более крупный. Обозначить конец сценария в этом случае можно с помощью маркера `__END__`.

Используете вы `-x` или нет, но при синтаксическом анализе в строке `#!` всегда выполняется поиск ключей. Благодаря этому при работе на платформе, поддерживающей только один аргумент в строке `#!` или, того хуже, даже не воспринимающей строку `#!` как особую, можно иметь единообразное восприятие ключей независимо от того, как был вызван Perl, даже если для поиска начала сценария был указан ключ `-x`.

Предупреждение: поскольку в ранних версиях Unix ядро молча заканчивает интерпретацию строки `#!` после 32 символов, некоторые ключи программа может получить в целости, а другие – нет; можно даже получить «`-`» без последующей буквы, если не проявить осторожность. Возможно, потребуется сделать так, чтобы все ключи находились по одну сторону (до или после) этой границы в 32 символа. Для большинства ключей неважно, если они обработаются лишний раз, но получение «`-`» вместо целого ключа приведет к тому, что Perl будет пытаться прочесть исходный код со стандартного ввода, а не из вашего сценария. Неполный ключ `-I` тоже может привести к неожиданным результатам. Однако некоторые ключи не безразличны к повторной обработке, например комбинации `-l` и `-0`. Либо поместите все ключи за 32-символьной границей (если это возможно), либо вместо `-0`*`DIGITS`* используйте `BEGIN{ $/ = "\0`*`DIGITS`*`"; }`. Конечно, если вы работаете не под Unix, то с такой проблемой вы не столкнетесь.

Синтаксический разбор ключей в строке `#!` начинается с того места, где впервые встретится «perl». Последовательности «`-*`» и «`- `» специально игнорируются в интересах пользователей *emacs*, поэтому при желании можно записать:

```
#!/bin/sh - - # -*- perl -*- -p
eval 'exec perl -S $0 ${1+"$@"}'
    if 0;
```

и Perl распознает только ключ `-p`. Причудливое сочетание «`-*- perl -*-`» сообщает *emacs* о необходимости начать работу в режиме Perl; если вы не пользуетесь *emacs*, оно вам не нужно. Путаница с `-S` разъясняется ниже в описании этого ключа.

Аналогичный трюк связан с программой *env*(1), если она у вас есть:

```
#!/usr/bin/env perl
```

В предыдущих примерах используется относительный путь к интерпретатору Perl, находящий ту его версию, которая первой встретится в маршруте поиска пользователя. Если требуется особая версия Perl, например *perl5.6.1*, поместите ее прямо в путь строки `#!`, используете ли вы программу *env*, ключ `-S` или обычную обработку `#!`.

Если строка `#!` *не* содержит слово «perl», то вместо интерпретатора Perl выполняется программа, указанная после `#!`. Предположим, например, что у вас есть обычный сценарий оболочки Борна с текстом:

```
#!/bin/sh
echo "I am a shell script"
```

Если подать этот файл на вход Perl, то Perl запустит */bin/sh*. Это несколько странно, но полезно на машинах, которые не распознают `#!`, поскольку путем установки переменной окружения `SHELL` можно сообщить программе (например, почтовой), что оболочкой служит */usr/bin/perl*, и Perl отправит программу нужному интерпретатору, даже если ядро настолько глупо, что не может этого сделать само.

Но вернемся к сценариям Perl, действительно являющимся сценариями Perl. Отыскав ваш сценарий, Perl компилирует всю программу во внутреннее представление (см. главу 18 «Компиляция»). При возникновении ошибок компиляции выполнение не начинается. (В этом отличие от обычных сценариев оболочки и командных файлов, которые могут частично выполниться, прежде чем будет обнаружена синтаксическая ошибка.) Если сценарий синтаксически корректен, он выполняется. Если сценарий выполнен до конца и при этом не встретились операторы `exit` или `die`, Perl выполняет неявный `exit(0)`, сообщая вызывающей программе об успешном выполнении. (Это поведение отличается от обычных программ C, в которых в качестве кода завершения можно получить случайное число, если программа завершается обычным образом.)

#! и кавычки в системах, отличных от Unix

Механизм `#!`, используемый в Unix, можно моделировать на других системах:

Macintosh

Программа Perl на Macintosh должна иметь соответствующие Creator и Type, чтобы приложение Perl вызывалось двойным щелчком.

MS-DOS

Создайте пакетный файл для запуска вашей программы и закодируйте его согласно `ALTERNATIVE_SHEBANG`. Дополнительные сведения можно получить из файла *dosish.h*, находящегося в корневом каталоге дистрибутива исходного кода Perl.

OS/2

Поместите строку:

```
extproc perl -S -your_switches
```

первой в файле **.cmd* (`-S` позволяет обойти ошибку обработки `"extproc"` в *cmd.exe*).

VMS

Поместите в начале программы строки:

```
% perl -mysw 'f$env("procedure")' 'p1' 'p2' 'p3' 'p4' 'p5' 'p6' 'p7' 'p8' !
$ exit++ + ++$status != 0 and $exit = $status = undef;
```

где *-mysw* обозначает ключи командной строки, которые нужно передать Perl. После этого можно вызвать программу как процедуру DCL, непосредственно введя с клавиатуры `perl program` и сказав `@program`, или неявным образом через `DCL$PATH`, используя лишь имя программы. Это «заклинание» немного сложно запомнить, но Perl отобразит его, если вы введете `perl "-V:startperl"`. Если вам и этого не запомнить...– что ж, для того вы и купили эту книгу.

Win??

При использовании дистрибутива Perl ActiveState с какой-либо разновидностью операционной системы Microsoft Windows (а именно Win95, Win98, Win00,[1] WinNT, но не Win3.1), процедура установки Perl модифицирует реестр Windows, связывая расширение *.pl* с интерпретатором Perl.

При установке другого варианта Perl, содержащегося в каталоге Win32 дистрибутива Perl, нужно модифицировать реестр Windows вручную.

Обратите внимание на то, что использование расширения *.pl* означает, что больше нельзя отличить исполняемую программу Perl от файла «библиотеки perl». Чтобы избежать этого, можно использовать для программ Perl расширение *.plx*. В последнее время масштабы этого бедствия уменьшились, поскольку большинство модулей Perl теперь имеет расширение *.pm*.

Интерпретаторы команд на системах, отличных от Unix, часто имеют совершенно другие представления о применении кавычек. Необходимо знать, какие специальные символы использует интерпретатор команд (обычно это `*`, `\` и `"`) и как представить пробельные символы и специальные символы, чтобы запускать однострочные программы посредством ключа `-e`. Может также потребоваться заменить одиночный `%` на `%%` или преобразовать его другим образом, если это специальный символ вашей оболочки.

В некоторых системах может потребоваться заменить одиночные кавычки двойными. Только не делайте этого в системах Unix или Plan9, а также в оболочках в стиле Unix, таких как системы MKS Toolkit или из пакета Cygwin, производимого в Cygnus, входящей теперь в Redhat. Новый эмулятор Unix под названием Interix, разработанный Microsoft, тоже начинает выглядеть э-э... интериксно.

Например, в Unix и Mac OS X командная строка будет выглядеть так:

```
% perl -e 'print "Hello world\n"'
```

На Macintosh (до Mac OS X) так:

```
print "Hello world\n"
```

после чего надо выполнить «Myscript» или нажать клавиши <Shift>+<Command>+<R>.

---

[1] Приносим извинения за технические неполадки...

Вид командной строки в VMS:

```
$ perl -e "print ""Hello world\n"""
```

или с qq//:

```
$ perl -e "print qq(Hello world\n)"
```

А в MS-DOS и прочих:

```
A:> perl -e "print \"Hello world\n\""
```

или qq// для выбора собственных кавычек:

```
A:> perl -e "print qq(Hello world\n)"
```

Проблема в том, что ни один из этих способов не надежен: он зависит от используемого процессора команд. Если командной оболочкой служит *4DOS*, лучше может работать следующее:

```
perl -e "print <Ctrl-x>"Hello world\n<Ctrl-x>""
```

Программа *CMD.EXE* из Windows NT, похоже, незаметно для всех отхватила значительную часть стандартных функций оболочки Unix, но пусть кто-нибудь попробует найти документацию по действующим в ней правилам применения кавычек.

На Macintosh[1] все зависит от используемой среды. Оболочка MacPerl, или MPW, способом, весьма схожим с оболочками Unix, поддерживает несколько вариантов применения кавычек, но позволяет свободно использовать в качестве управляющих не-ASCII-символы Macintosh.

Общего решения здесь нет, и царит здесь полнейшая неразбериха. Если вы работаете в системе, отличной от Unix, но вам нравится командная строка, постарайтесь приобрести интерпретатор команд получше того, который вам изначально поставлен, что не должно оказаться сложным делом.

Или пишите все на Perl и забудьте об однострочниках.

Местонахождение Perl

Это может казаться очевидным, но Perl полезен только тогда, когда пользователь легко может найти его. Если это возможно, то хорошо, чтобы */usr/bin/perl* и */usr/local/bin/perl* были символическими ссылками на фактический исполняемый модуль. Если это невозможно, то системному администратору настоятельно рекомендуется поместить Perl и сопутствующие утилиты в каталог, входящий в стандартный PATH пользователя или другое удобное место.

---

[1] По крайней мере до выхода Mac OS X, которая, что весьма удачно, происходит от BSD.

В данной книге мы прибегаем к стандартному обозначению `#!/usr/bin/perl` в первой строке программы независимо от конкретного механизма, действующего в системе. Для того чтобы запустить специфичную версию Perl, используйте специальный путь:

```
#!/usr/local/bin/perl5.6.0
```

Если вы хотите запустить версию *не ниже* некоторого номера, но не возражаете против более высоких номеров, поместите такую команду в верхней части программы:

```
use v5.6.0;
```

(Замечание: ранние версии Perl используют такие номера, как 5.005 или 5.004_05. Сегодня мы считали бы их версиями 5.5.0 и 5.4.5, но версии Perl старше 5.6.0 не понимают таких обозначений.)

Ключи

Ключ командной строки из одного символа без собственного аргумента всегда можно связать с ключом, который следует за ним.

```
#!/usr/bin/perl -spi.bak    # same as -s -p -i.bak
```

Ключи (switches) называют также опциями (options) или флагами (flags). Как бы вы их ни называли, вот какие из них Perl может распознать:

`- -`

Прекращает обработку ключей, даже если следующий аргумент начинается со знака «минус». Другого действия не оказывает.

`-0`*`OCTNUM`*

`-0` Задает разделитель записей (`$/`) в виде восьмеричного числа. Если *`OCTNUM`* отсутствует, разделителем становится символ NUL (т. е. ASCII-символ 0, `"\0"` в Perl). До или после восьмеричного числа могут находиться другие ключи. Например, если у вас есть версия *find*(1), способная выводить имена файлов, оканчивающиеся нулевым символом, можно написать так:

```
% find . -name '*.bak' -print0 | perl -n0e unlink
```

Особое значение `00` заставляет Perl читать файлы в режиме параграфов, что эквивалентно установке переменной `$/` равной `""`. А значение `0777` – проглатывать сразу целые файлы. Это эквивалентно тому, что переменная `$/` не определена. Мы используем `0777`, поскольку символа ASCII с таким значением нет. (К несчастью, символ Unicode с таким значением *есть*: `\N{LATIN SMALL LETTER O WITH STROKE AND ACUTE}` – маленькое латинское «о» со штрихом и акутом, но что-то подсказывает нам, что вы не станете использовать его в качестве разделителя своих записей.)

-a Включает режим авторасщепления, но указывается только совместно с -n или -p. В неявном цикле while, организуемом ключами -n и -p, прежде всего неявно выполняется команда split в массив @F. Поэтому

```
% perl -ane 'print pop(@F), "\n";'
```

эквивалентно:

```
LINE: while (<>) {
    @F = split(' ');
    print pop(@F), "\n";
}
```

Можно задать новый разделитель полей, передав в split ключ -F. Например, эквивалентны следующие два вызова:

```
% awk -F: '$7 && $7 !~ /^\/bin/' /etc/passwd
% perl -F: -lane 'print if $F[6] && $F[6] !~ m(^/bin)' /etc/passwd
```

-c Заставляет Perl проверить синтаксис сценария, а затем выйти, не выполняя того, что скомпилировано. Технически ключ делает несколько большее: выполняются все блоки BEGIN и CHECK, а также директивы use, поскольку считается, что они обрабатываются до выполнения программы. Однако блоки INIT и END не выполняются. Прежний, но редко требуемый режим можно все же получить, поместив в конце основного сценария:

```
BEGIN { $^C = 0; exit; }
```

-C Позволяет Perl использовать на целевой системе API с родными расширенными символами, если они поддерживаются (что касается версии 5.6.0, то он работает только на платформах Microsoft). Специальная переменная ${^WIDE_SYSTEM_CALLS} отражает состояние этого ключа.

-d Запускает сценарий в отладчике Perl. См. главу 20 «Отладчик Perl».

-d:*MODULE*

Запускает сценарий под управлением отладочного или трассировочного модуля, установленного в библиотеке Perl как Devel::*MODULE*. Например, -d:DProf выполняет сценарий с использованием профайлера Devel::DProf. См. также раздел об отладке в главе 20.

-D*LETTERS*

-D*NUMBER*

Устанавливает флаги отладки. (Работает, только если ваша версия Perl компилирована с поддержкой отладки, как описано ниже.) Можно задать либо число, представляющее собой сумму нужных битов, либо список букв. Например, чтобы посмотреть, как выполняется ваш сценарий, задайте -D14 или -Dslt. Другим полезным значением является -D1024 или -Dx, выводящее скомпилированное синтаксическое дерево. Ключ -D512 или -Dr выводит скомпилированные регулярные выражения. Численное

значение внутренне доступно как специальная переменная `$^D`. Значения битов перечислены в табл. 19.1.

Таблица 19.1. Опции `-D`

Бит	Буква	Значение
1	p	Выводить результаты лексического и синтаксического разбора
2	s	Выводить состояние стеков
4	l	Обработка стека меток
8	t	Трассировка выполнения
16	o	Разрешение методов и перегрузки
32	c	Преобразования строка/число
64	P	Препроцессор печати для `-P`
128	m	Выделение памяти
256	f	Обработка форматов
512	r	Разбор и выполнение регулярных выражений
1024	x	Дамп дерева синтаксического разбора
2048	u	Проверка меченых данных
4096	L	Поверка утечки памяти (требует ключа `-DLEAKTEST` при компиляции Perl)
8192	H	Дамп хешей – узурпирует `values()`
16384	X	Выделение временной памяти
32768	D	Сборка мсора
65536	S	Синхронизация потоков

Для всех этих флагов нужен исполняемый модуль Perl, специально компилированный для поддержки отладки. По умолчанию Perl компилируется без поддержки отладки, поэтому вы не сможете использовать ключ `-D`, пока сами не скомпилируете эту версию Perl для отладки или это не сделает ваш системный администратор. Подробности можно найти в файле *INSTALL* в каталоге исходного кода Perl, но вкратце сущность состоит в том, чтобы передать компилятору C ключ `-DDEBUGGING` при компиляции Perl. Этот флаг автоматически устанавливается, если включить опцию `-g`, когда *Configure* запрашивает у пользователя флаги оптимизатора и отладчика.

Если вы просто хотите получить распечатку каждой строки программы Perl по ходу ее выполнения (аналогично тому, что дает *sh -x* для сценариев оболочки), ключ Perl `-D` не поможет. Вместо этого нужно сделать следующее:

```
# Синтаксис оболочки Борна (bash)
$ PERLDB_OPTS="NonStop=1 AutoTrace=1 frame=2" perl -dS program
# Синтаксис csh
% (setenv PERLDB_OPTS "NonStop=1 AutoTrace=1 frame=2"; perl -dS program)
```

За подробностями и вариантами обратитесь к главе 20.

-e *PERLCODE*

Может использоваться для ввода одной или более строк сценария. Если указан ключ -e, Perl не ищет имя файла программы в списке аргументов. Аргумент *PERLCODE* рассматривается, как если бы он заканчивался символом новой строки, поэтому можно дать несколько команд -e, чтобы создать программу из нескольких строк. (Не забудьте использовать точку с запятой там, где вы должны это делать в обычной программе, сохраняемой в файле.) То, что -e снабжает каждый аргумент символом новой строки, не значит, что вы обязаны использовать несколько ключей -e, если ваша оболочка поддерживает заключение в кавычки нескольких строк, как это делают *sh*, *ksh* и *bash*, то можно передать сценарий из нескольких строк в одном аргументе -e:

```
$ perl -e 'print "Howdy, ";
           print "@ARGV!\n";' world
Howdy, world!
```

В *csh* лучше задать несколько ключей -e:

```
% perl -e 'print "Howdy, ";' \
        -e 'print "@ARGV!\n";' world
Howdy, world!
```

Учитываются как явные, так и неявные символы новой строки, поэтому в обоих случаях вторая команда print находится на строке 2 сценария -e.

-F*PATTERN*

Задает шаблон для разбивки, когда ключ -a включает авторазбивку (в противном случае не оказывает никакого воздействия). Шаблон может быть заключен между символами косой черты (//), двойных кавычек (""") или одинарных кавычек (' '). В противном случае он автоматически заключается в одинарные кавычки. Не забывайте о том, что для передачи кавычек через оболочку сами они должны быть закавычены способом, который зависит от используемой оболочки.

-h Выводит сводку ключей командной строки Perl.

-i*EXTENSION*

-i Указывает, что файлы, обрабатываемые посредством конструкции <>, должны редактироваться «по месту». Это производится путем переименования входного файла, открытия выходного файла с исходным именем и выбора этого выходного файла в качестве файла по умолчанию для вызовов print, printf и write.[1]

Аргумент *EXTENSION* используется для изменения имени прежнего файла с целью создания резервной копии. Если *EXTENSION* не задан, резервная копия не создается и перезаписывается текущий файл. Если *EXTENSION* не

[1] Технически это не совсем «по месту». Имя файла то же, но физически это другой файл.

содержит символ *, то заданная строка дописывается в конец текущего имени файла. Если *EXTENSION* содержит один или несколько символов *, то каждый символ * заменяется именем обрабатываемого в данный момент файла. На языке Perl это можно выразить как:

```
($backup = $extension) =~ s/\*/$file_name/g;
```

Благодаря этому можно использовать для резервного файла префикс вместо или даже в дополнение к суффиксу:

```
% perl -pi'orig_*' -e 's/foo/bar/' xyx     # создать копию с именем 'orig_xyx'
```

Можно даже поместить резервные копии исходных файлов в другой каталог (при условии, что он уже существует):

```
% perl -pi'old/*.orig' -e 's/foo/bar/' xyx # создать копию 'old/xyx.orig'
```

Следующие пары однострочных команд эквивалентны:

```
% perl -pi -e 's/foo/bar/' xyx             # перезаписать текущий файл
% perl -pi'*' -e 's/foo/bar/' xyx           # перезаписать текущий файл
% perl -pi'.orig' -e 's/foo/bar/' xyx     # сделать резервную копию в 'xyx.orig'
% perl -pi'*.orig' -e 's/foo/bar/' xyx    # сделать резервную копию в 'xyx.orig'
```

В оболочке сказать:

```
% perl -p -i.orig -e "s/foo/bar/;"
```

означает то же самое, что использовать программу:

```
#!/usr/bin/perl -pi.orig
s/foo/bar/;
```

что служит удобной сокращенной записью для существенно более длинной:

```
#!/usr/bin/perl
$extension = '.orig';
LINE: while (<>) {
    if ($ARGV ne $oldargv) {
        if ($extension !~ /\*/) {
            $backup = $ARGV . $extension;
        }
        else {
            ($backup = $extension) =~ s/\*/$ARGV/g;
        }
        unless (rename($ARGV, $backup)) {
            warn "cannot rename $ARGV to $backup: $!\n";
            close ARGV;
            next;
        }
        open(ARGVOUT, ">$ARGV");
        select(ARGVOUT);
        $oldargv = $ARGV;
```

```
        }
        s/foo/bar/;
    }
    continue {
        print;  # вывод в соответствии с исходным именем файла
    }
    select(STDOUT);
```

Этот длинный код в сущности идентичен простому однострочному с ключом -i, за вычетом того, что в форме -i не требуется сравнивать $ARGV с $oldargv, чтобы узнать о том, что имя изменилось. Однако в нем используется ARGVOUT для выбранного дескриптора файла и восстанавливается STDOUT как дескриптор файла вывода по умолчанию после выполнения цикла. Как и в приведенном коде, Perl создает резервную копию файла вне зависимости от того, изменился ли в действительности вывод. В описании функции eof можно найти примеры использования eof без скобок для нахождения конца каждого входного файла в случае, если требуется дописывать в конец каждого файла или сбрасывать нумерацию строк.

Если Perl не может создать резервную копию данного файла, как указано в *EXTENSION*, то выводится предупреждающее сообщение и продолжается обработка остальных перечисленных файлов.

Нельзя применять -i для создания каталогов или удаления расширений из файлов. Нельзя также использовать его с тильдой ~ для указания исходного каталога, что и хорошо, поскольку некоторые любят включать этот символ в имена файлов резервных копий:

```
% perl -pi~ -e 's/foo/bar/' file1 file2 file3...
```

Наконец, ключ -i не прекращает выполнение Perl, если в командной строке не заданы имена файлов. В этом случае не делаются резервные копии, поскольку нельзя определить исходный файл, и обработка продолжается, как можно было ожидать, с STDIN на STDOUT.

-I*DIRECTORY*

Каталоги, задаваемые с помощью -I, присоединяются спереди к @INC, в котором содержатся пути поиска модулей. -I сообщает также препроцессору C, где искать включаемые файлы. Препроцессор C вызывается ключом -P; по умолчанию он осуществляет поиск в */usr/include* и */usr/lib/perl*. Если только вы не собираетесь работать с препроцессором C (а почти никто из программистов этого больше не делает), лучше использовать в сценарии директиву use lib. Однако, как и use lib, ключ -I неявно добавляет каталоги, специфические для платформы. Подробности о директиве use lib см. в главе 31 «Модули прагм».

-l*OCTNUM*

-l Включает автоматическую обработку конца строки. Его применение дает два результата: во-первых, автоматически съедается символ окончания строки при использовании с ключами -n или -p, а во-вторых, в $\ записы-

вается значение *OCTNUM*, в результате чего все команды print получают дополнительно символ окончания строки со значением ASCII, равным *OCTNUM*. Если *OCTNUM* опустить, то -l устанавливает переменную $\ равной текущему значению $/, обычно – символу новой строки. Поэтому, чтобы ограничить строки 80-ю колонками, введите следующее:

```
% perl -lpe 'substr($_, 80) = ""'
```

Обратите внимание, что присвоение $\ = $/ производится при обработке ключа, поэтому разделитель входных записей может не совпадать с разделителем выходных записей, если за ключом -l следует ключ -0:

```
% gnufind / -print0 | perl -ln0e 'print "found $_" if -p'
```

В этой команде $\ устанавливается равной символу новой строки, а затем $/ устанавливается равной нулевому символу. (Обратите внимание, что ключ 0 интерпретировался бы как часть ключа -l, если бы непосредственно следовал за ним, поэтому мы вставили между ними ключ -n.)

-m и -M

Эти ключи загружают *MODULE*, как если бы вы выполнили use, если только не задать *-MODULE* вместо *MODULE*, и тогда вызывается no. Например, -Mstrict аналогично use strict, а -M-strict аналогично no strict.

-m*MODULE*

Выполняет use *MODULE* () перед выполнением вашего сценария.

-M*MODULE*

-M'*MODULE...*'

Выполняет use *MODULE*, перед тем как запустить ваш сценарий. Команда образуется простой интерполяцией оставшейся части аргумента после -M, поэтому можно использовать кавычки, чтобы добавить код после имени модуля, например -M'MODULE qw(foo bar)'.

-M*MODULE*=arg1, arg2...

Маленькое синтаксическое удобство, позволяющее указать -Mmodule=foo,bar в качестве сокращения для -M'module qw(foo bar)'. Это позволяет избежать применения кавычек при импортировании символов. Фактически -Mmodule=foo,bar генерирует следующий код:

```
use module split(/,/, q{foo,bar})
```

Важно учитывать, что в формате = пропадает различие между -m и -M, но во избежание путаницы следует предпочесть написание в верхнем регистре.

Ключи -M и -m можно использовать только при действительном вызове Perl из командной строки, а не как параметры, указанные в строке #!. (Если вам нужно вставить их в файл, почему вместо этого просто не написать эквивалентные use или no ?)

-n Заставляет Perl выполнить ваш сценарий в следующем цикле, который повторяется с аргументами имен файлов, подобно тому как это делают *sed -n* и *awk*:

```
LINE:
while (<>) {
     ...                # здесь находится ваш сценарий
}
```

В сценарии LINE может играть роль метки цикла, несмотря на то, что фактически метки в файле не видно.

Обратите внимание: по умолчанию строки не выводятся. О выводе строк см. описание -p. Вот эффективный способ удалить все файлы, созданные больше недели назад:

```
find . -mtime +7 -print | perl -nle unlink
```

Это выполняется быстрее, чем при использовании ключа *-exec* в *find*(1), поскольку не приходится запускать процесс для каждого найденного имени файла. Поразительное совпадение: блоки BEGIN позволяют организовать перехват управления до или после неявного цикла, совсем как в *awk*.

-p Заставляет Perl выполнить ваш сценарий в следующем цикле, который повторяется с аргументами имен файлов, подобно тому как это делает *sed*:

```
LINE:
while (<>) {
     ...         # здесь находится ваш сценарий
}
continue {
     print or die "-p destination: $!\n";
}
```

Можно использовать LINE в сценарии как метку цикла, несмотря на то, что фактически метки в файле не видно.

Если по какой-либо причине файл, заданный в аргументе, нельзя открыть, Perl выводит предупреждающее сообщение и переходит к следующему файлу. Обратите внимание, что строки выводятся автоматически. Ошибка, возникающая во время вывода, считается фатальной. Еще одно поразительное совпадение: блоки BEGIN и END можно использовать для перехвата управления до или после неявного цикла, совсем как в *awk*.

-P Вызывает обработку сценария препроцессором C перед компиляцией Perl. (Поскольку и комментарии и директивы *cpp*(1) начинаются с символа #, следует избегать в начале комментариев слов, которые распознаются препроцессором C, таких как «if», «else» или «define».) Независимо от использования ключа -P Perl выполняет директивы #line, управляющие номерами строк и именем файла, поэтому любой препроцессор может известить Perl об этих вещах. См. раздел «Генерация Perl из других языков» главы 24 «Распространенные приемы программирования».

-s Разрешает элементарный анализ ключей, находящихся в командной строке после имени сценария, но перед аргументами имени файла или завершающим обработку ключом «--». Все найденные ключи удаляются из @ARGV, а в Perl устанавливается переменная с таким же именем, как у ключа. Совмещение ключей не разрешается, поскольку допускаются ключи, состоящие из нескольких символов.

В следующем сценарии "true" выводится только в случае запуска сценария с ключом -foo.

```
#!/usr/bin/perl -s
if ($foo) { print "true\n" }
```

Если ключ имеет вид -xxx=yyy, то переменная $xxx устанавливается равной тому значению, которое следует в аргументе за знаком равенства (в данном случае "yyy "). В следующем сценарии "true" выводится в том и только том случае, когда сценарий вызывается с ключом -foo=bar.

```
#!/usr/bin/perl -s
if ($foo eq 'bar') { print "true\n" }
```

-S Заставляет Perl использовать при поиске сценария переменную окружения PATH (если имя сценария не содержит символа, разделяющего каталоги).

Обычно этот ключ применяется для облегчения эмуляции запуска посредством #! на платформах, не поддерживающих #!. На многих платформах, имеющих оболочку, совместимую с оболочками Борна или C, можно использовать такую команду:

```
#!/usr/bin/perl
eval "exec /usr/bin/perl -S $0 $*"
        if $running_under_some_shell;
```

Система игнорирует первую строку и передает сценарий оболочке */bin/sh*, которая пытается выполнить сценарий Perl как сценарий оболочки. Оболочка выполняет вторую строку как обычную команду оболочки и таким образом запускает интерпретатор Perl. В некоторых системах $0 может не содержать полный путь, поэтому -S сообщает Perl, что при необходимости он должен найти сценарий. После того как Perl находит сценарий, он производит синтаксический анализ строк и игнорирует их, поскольку переменная $running_under_some_shell не равна true. Лучшей конструкцией, чем $*, была бы ${1+"$@"}, которая обрабатывает содержащиеся в именах файлов пробелы и т. п., но не работает, если сценарий интерпретируется как *csh-сценарий*. Для того чтобы запустить *sh*, а не *csh*, некоторые системы должны заменить строку #! строкой, содержащей просто двоеточие, которую Perl вежливо проигнорирует. Другие системы не могут это контролировать и требуют совершенно окольной (полностью универсальной) конструкции, которая будет работать как в *csh*, *sh*, так и *perl*, например такой:

```
eval '(exit $?0)' && eval 'exec /usr/bin/perl -S $0 ${1+"$@"}'
     & eval 'exec /usr/bin/perl -S $0 $argv:q' if 0;
```

Конечно, смотрится это безобразно, но таковы системы, работающие[1] подобным образом.

На некоторых платформах ключ `-S` заставляет также Perl добавлять суффиксы к имени файла, который он ищет. Например, на платформах Win32 добавляются суффиксы *.bat* и *.cmd*, когда поиск исходного имени оказывается неудачным, а имя не оканчивается одним из этих суффиксов. Если Perl скомпилирован с поддержкой отладки, то можно посмотреть, как происходит поиск, с помощью ключа Perl `-Dp`.

Если переданное имя файла содержит разделители имен каталогов (даже если это относительный, а не абсолютный путь) и файл не найден, платформы, неявно добавляющие расширения имен файлов (не Unix), сделают это и будут поочередно искать файлы с этими добавленными расширениями.

На DOS-образных платформах, если сценарий не содержит разделителей имен каталогов, его поиск сначала осуществляется в текущем каталоге, а потом согласно переменной `PATH`. На платформах Unix поиск сценария осуществляется строго по `PATH` из соображений безопасности, требующих предотвращения случайного запуска программы из текущего каталога, когда явного запроса на это нет.

`-T` Включает проверки «меченых» данных («taint» checks), которые становится возможным использовать. Обычно эти проверки осуществляются только при выполнении с установленными setuid или setgid. Неплохо явно включить их для программ, выполняемых от другого имени, таких как программы CGI. См. главу 23 «Защита данных».

Обратите внимание, что по соображениям безопасности Perl должен увидеть этот параметр как можно раньше; обычно это значит, что он должен стоять в начале командной строки или строки `#!`. Если он расположен недостаточно близко к началу, то Perl выражает неудовольствие.

`-u` Заставляет Perl выполнять дамп памяти после компиляции сценария. Теоретически можно взять этот дамп и превратить в исполняемый модуль с помощью программы *undump* (она не поставляется). При этом запуск ускоряется ценой некоторого расхода дискового пространства (его можно минимизировать путем усечения (stripping) выполняемого модуля). Если вы хотите сначала выполнить часть сценария, а потом – дамп, используйте вместо этого ключа оператор Perl `dump`. Замечание: наличие программы *undump* зависит от платформы; в некоторых портированных версиях Perl не на Unix-платформах она может отсутствовать. Она заменена новым генератором кода из Perl в C, который отличается лучшей переносимостью (но все еще является опытным).

`-U` Позволяет Perl осуществлять небезопасные операции. В данное время «небезопасными» являются только операции удаления каталогов при работе с правами суперпользователя и запуск программ с setuid, сопровождаю-

[1] Термин употреблен нами после тщательного рассмотрения.

щиеся преобразованием фатальных проверок меченых данных в предупреждения. Учтите, что вывод предупреждений должен быть включен для того, чтобы действительно выдавались предупреждения о меченых данных.

-v Выводит номер версии и уровень патчей выполняемого модуля Perl, а также некоторые дополнительные сведения.

-V Выводит сводку основных параметров конфигурации Perl и текущее значение `@INC`.

-V:*NAME*

Выводит на `STDOUT` значение указанной переменной конфигурации. *NAME* может содержать символы из регулярных выражений, такие как «`.`» для соответствия любому символу и «`.*`» для соответствия произвольной последовательности символов.

```
% perl -V:man.dir
man1dir='/usr/local/man/man1'
man3dir='/usr/local/man/man3'

% perl -V:'.*threads'
d_oldpthreads='undef'
use5005threads='define'
useithreads='undef'
usethreads='define'
```

При запросе для несуществующей переменной конфигурации ее значение сообщается как `"UNKNOWN"`. Внутри программы сведения о конфигурации можно получить с помощью модуля `Config`, однако он не поддерживает шаблоны в индексах хешей:

```
% perl -MConfig -le 'print $Config{man1dir}'
/usr/local/man/man1
```

См. обсуждение модуля `Config` в главе 32 «Стандартные модули».

-w Выводит предупреждения о переменных, которые встретились единственный раз, и скалярных величинах, используемых до присвоения им значений. Также выводятся предупреждения о переопределении подпрограмм и ссылках на неопределенные дескрипторы файлов или дескрипторы файлов, открытых только для чтения, но в которые предпринимается попытка записи. Выводятся также предупреждения об использовании как чисел величин, не похожих на числа, об использовании массива как скаляра, рекурсии в подпрограммах глубиной выше 100 и массе других вещей. Это все сообщения, отмеченные символами «(W)» в главе 33 «Диагностические сообщения».

Этот ключ просто устанавливает глобальную переменную `$^W`. Он не оказывает влияния на лексические предупреждения – см. описание ключей **-W** и **-X**. Включать или отключать отдельные сообщения можно с помощью прагмы `use warnings`, описанной в главе 31.

-W Безусловно и постоянно включает вывод всех предупреждений в программе, даже если они отключены локально с помощью `no warnings` или `$^W = 0`. Распространяется на все файлы, загруженные с помощью `use`, `require` или `do`. Можете рассматривать этот ключ как Perl-эквивалент команды *lint*(1).

-x*DIRECTORY*

-x Требует от Perl извлечь сценарий, встроенный в сообщение. Весь мусор, находящийся впереди, отбрасывается, пока не будет найдена строчка, начинающаяся с `#!` и содержащая слово `"perl"`. Все поддающиеся интерпретации ключи в этой строке после слова `"perl"` будут применены. Если задано имя каталога, то Perl перейдет в него перед запуском сценария. Ключ **-x** управляет удалением только мусора, находящегося в начале, но не в конце. Сценарий должен заканчиваться `__END__` или `__DATA__`, если в конце есть мусор, который нужно проигнорировать. (При желании сценарий может обработать весь мусор или его часть, находящуюся в конце, с помощью дескриптора файла `DATA`. Теоретически можно даже найти начало файла и обработать мусор, находящийся впереди.)

-X Безоговорочно и постоянно отключает вывод всех предупредительных сообщений – в точности обратно тому, что делает флаг **-W**.

Переменные окружения

В дополнение к различным ключам, явным образом изменяющим режим работы Perl, можно устанавливать переменные окружения, влияющие на различные основные режимы. Способ установки этих переменных окружения зависит от системы, однако есть один прием, о котором нужно знать, если вы используете *sh*, *ksh* или *bash*, состоящий в том, что можно временно устанавливать переменную окружения для одной команды, как если бы она была своеобразным ключом. Эта переменная должна быть установлена впереди команды:

```
$ PATH='/bin:/usr/bin' perl myproggie
```

Нечто аналогичное можно делать в подоболочке для *csh* и *tcsh*:

```
% (setenv PATH "/bin:/usr/bin"; perl myproggie)
```

В других случаях переменные окружения обычно устанавливаются в некотором файле с именем типа *.chsrc* или *.profile*, находящемся в исходном каталоге. В *csh* и *tcsh* можно сказать:

```
% setenv PATH '/bin:/usr/bin'
```

А в *sh*, *ksh* и *bash* команда будет выглядеть так:

```
$ PATH='/bin:/usr/bin'; export PATH
```

В других системах есть свои способы установки этих переменных на полупостоянной основе. Вот несколько переменных окружения, которые распознает Perl:

`HOME`

Используется, когда `chdir` вызывается без аргумента.

`LC_ALL, LC_CTYPE, LC_COLLATE, LC_NUMERIC, PERL_BADLANG`

Переменные окружения, управляющие тем, как Perl обрабатывает данные, характерные для отдельных естественных языков. См. электронную документацию по *perllocale*.

`LOGDIR`

К этой переменной Perl обращается, когда `chdir` вызывается без аргументов, но переменная `HOME` не установлена.

`PATH`

Используется при выполнении подпроцессов и для поиска программ, когда задан ключ `-S`.

`PERL5LIB`

Разделяемый двоеточиями список каталогов, в которых нужно искать библиотечные файлы Perl, прежде чем заглядывать в стандартный каталог библиотек и текущий каталог. Если в указанных каталогах существуют специфические для данной архитектуры каталоги, то они автоматически включаются в путь поиска. В целях обеспечения обратной совместимости с более старыми версиями при не определенной переменной `PERL5LIB` осуществляется обращение к `PERLLIB`.

При проверке меченых данных (из-за запуска в программе setuid или setgid либо указания ключа `-T`) переменные, задающие поиск библиотек, не используются. В такие программы необходимо включать с этой целью директиву `use lib`.

`PERL5OPT`

Ключи командной строки, принимаемые по умолчанию. Ключи из этой переменной используются так, как если бы они присутствовали в каждой командной строке Perl. Допустимы только ключи `-[DIMUdmw]`. При проверке меченых данных (либо при запуске в программе setuid или setgid, либо в случае указания ключа `-T`) эта переменная игнорируется. Если `PERL5OPT` начинается с `-T`, включается проверка меченых данных, а все последующие ключи игнорируются.

`PERL5DB`

Команда предназначена для загрузки кода отладчика. По умолчанию:

```
BEGIN { require 'perl5db.pl' }
```

См. главу 20 для получения дополнительных сведений об использовании этой переменной.

PERL5SHELL (только портированные под Microsoft)

Может устанавливать альтернативный командный процессор (оболочку), который Perl должен использовать внутренне для выполнения команд через обратный слэш или system. По умолчанию это cmd.exe /x/c в WinNT и command.com /c в Win95. Предполагается, что в качестве разделителя в значении выступают пробелы. Каждому символу, который должен быть защищен (такому, как пробел или обратная косая черта), следует предпослать обратную косую черту.

Обратите внимание, что Perl не работает в этих целях с переменной COMSPEC, поскольку для нее пользователи находят весьма различные применения, что вызывает проблемы переносимости между платформами. Кроме того, если Perl будет работать с оболочкой, непригодной для интерактивной работы, то установка в COMSPEC такой оболочки может помешать правильному функционированию других программ (которые обычно ищут в COMSPEC оболочку, пригодную для интерактивного использования).

PERLLIB

Разделяемый двоеточиями список каталогов, в которых нужно искать библиотечные файлы Perl, прежде чем заглядывать в стандартный каталог библиотек и текущий каталог. Если определена PERL5LIB, то PERLLIB не используется.

PERL_DEBUG_MSTATS

Применяется, только если Perl скомпилирован при включении в дистрибутив функции malloc (т. е. если perl -V:d_mymalloc дает "define"). Если переменная установлена, то после выполнения возвращается статистика использования памяти. Если переменная установлена равной целому числу, превышающему единицу, статистика распределения памяти выводится и после компиляции.

PERL_DESTRUCT_LEVEL

Применяется, только если Perl скомпилирован с поддержкой отладки, и управляет режимом глобального удаления объектов и других ссылок.

Сам Perl не использует никакие другие переменные окружения, кроме тех, которые он должен сделать доступными выполняемой программе или запускаемым ею дочерним процессам. Другие модули, стандартные или нет, могут работать с другими переменными окружения. Например, прагма use re использует PERL_RE_TC и PERL_RE_COLORS, модуль Cwd использует PWD, а модуль CGI обращается к многочисленным переменным окружения, устанавливаемым демоном HTTP (т. е. веб-сервером) для передачи сценарию CGI.

Программы, выполняющие setuid, поступят правильно, если прежде чем что-либо делать, выполнят следующие строки – чтобы никого не обманывать:

```
$ENV{PATH}  = '/bin:/usr/bin';    # или что еще нужно
$ENV{SHELL} = '/bin/sh' if exists $ENV{SHELL};
delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};
```

Подробности см. в главе 23.

20

Отладчик Perl

Прежде всего, пытались ли вы пользоваться директивой `use warnings`?

Если вызвать Perl с ключом `-d`, то программа будет запущена через отладчик Perl. Он работает как интерактивная среда Perl, выдавая приглашение для ввода команд отладчика, которые позволяют просматривать исходный код, устанавливать контрольные точки, создавать дамп стека вызова функций, изменять значения переменных и т. д. Любая команда, которая не распознается отладчиком, непосредственно выполняется (при помощи `eval`) как код Perl внутри отлаживаемого в данный момент программного пакета. (Отладчик использует пакет `DB` для получения сведений о своем собственном состоянии, чтобы избежать помех отлаживаемой программе.) Это настолько удобно, что часто отладчик запускают только для того, чтобы интерактивно проверить действие конструкций Perl. В этом случае не важно, какая программа используется Perl для отладки, поэтому мы выберем не особенно содержательную:

```
% perl -de 42
```

В Perl отладчик не является программой, совершенно отдельной от той, которая отлаживается, как это обычно бывает в типичной среде программирования. Флаг `-d` сообщает компилятору о необходимости ввести данные исходного кода в деревья синтаксического разбора, которые он будет передавать интерпретатору. Это значит, что код должен быть сначала правильно скомпилирован, прежде чем с ним сможет работать отладчик. Если компиляция успешна, интерпретатор производит предварительную загрузку особого библиотечного файла Perl, содержащего сам отладчик.

```
% perl -d /path/to/program
```

Программа остановится непосредственно перед первой выполняемой командой этапа исполнения (о командах времени выполнения см. раздел «Использование отладчика» этой главы) и попросит ввести команду отладчика. Когда отладчик останавливается и показывает строку кода, то это та строка, которую он *собирается* выполнить, а не та, которая только что выполнена.

Добравшись до некоторой строки, отладчик сначала проверяет наличие контрольной точки, выводит ее (если находится в режиме трассировки), осуществляет действия (задаваемые командой, описанной далее в разделе «Команды отладчика») и, наконец, сообщает пользователю о наличии контрольной точки или о работе в пошаговом режиме. В противном случае он выполняет строку обычным образом и переходит к следующей.

Использование отладчика

Приглашение, которое выдает отладчик, выглядит примерно так:

```
DB<8>
```

или даже:

```
DB<<17>>
```

где число показывает количество выполненных команд. Механизм буфера команд в стиле *csh* позволяет вызывать предшествующие команды по номеру. Например, `!17` повторит команду с номером 17. Количество угловых скобок указывает на глубину вложенности отладки. Например, вы увидите более одной пары скобок, если находитесь в контрольной точке, а затем выводите результат вызова функции, в которой в самой установлена контрольная точка.

Для того чтобы войти в команду, состоящую из нескольких строк, такую как определение подпрограммы с несколькими командами, нужно перед переводом строки, который обычно завершает команду отладчика, поместить обратную косую черту. Вот пример:

```
DB<1> for (1..3) {         \
  cont:     print "ok\n";    \
  cont: }
  ok
  ok
  ok
```

Допустим, вы хотите запустить отладчик со своей программкой (назовем ее *camel_flea*) и остановиться, добравшись до функции с именем `infested`. Вот как это нужно сделать:

```
% perl -d camel_flea

Loading DB routines from perl5db.pl version 1.07
Editor support available.

Enter h or `h h' for help, or `man perldebug' for more help.
```

```
main::(camel_flea:2):     pests('bactrian', 4);
  DB<1>
```

Отладчик остановит программу прямо перед первой исполняемой командой этапа исполнения (см. ниже о командах времени компиляции) и попросит ввести команду. Еще раз напомним, что когда отладчик останавливается, чтобы показать строку кода, он показывает ту строку, которую *собирается* выполнить, а не ту, которая только что выполнена. Выведенная строка может выглядеть не совсем так, как в исходном файле, особенно если текст обработан препроцессором.

Теперь вам нужно остановиться, как только программа дойдет до функции `infested`, поэтому контрольная точка устанавливается следующим образом:

```
DB<1> b infested
DB<2> c
```

Теперь отладчик продолжает выполнение, пока не наткнется на эту функцию, и тогда сообщит следующее:

```
main::infested(camel_flea:8):       my $bugs = int rand(3);
```

Чтобы увидеть «окно» исходного кода, окружающего контрольную точку, воспользуйтесь командой `w`:

```
DB<2> w
5     }
6
7       sub infested {
8==>b       my $bugs = int rand(3);
9:          our $Master;
10:         contaminate($Master);
11:         warn "needs wash"
12              if $Master && $Master->isa("Human");
13
14:         print "got $bugs\n";
DB<2>
```

Как показывает маркер `==>`, текущая строка имеет номер 8, а символ `b` указывает, что на ней установлена контрольная точка. Если бы для нее было задано действие, то там также присутствовал бы символ `a`. На строках, номера которых имеют двоеточия, можно устанавливать прерывания, на остальных – нет.

Чтобы увидеть последовательность вызовов, попросите отладчик вывести стек с помощью команды `T`:

```
DB<2> T
$ = main::infested called from file `Ambulation.pm' line 4
@ = Ambulation::legs(1, 2, 3, 4) called from file `camel_flea' line 5
. = main::pests('bactrian', 4) called from file `camel_flea' line 2
```

Первый символ (`$`, `@`, или `.`) указывает на то, в каком контексте была вызвана функция – скалярном, списка или пустом, соответственно. Выведено три

строки, поскольку вы находились на глубине вызова трех функций, когда потребовали вывести стек вызовов. Ниже объясняется значение каждой из трех строк:

- Первая строка говорит о том, что вы находились в функции `main::infested`, когда запустили трассировку стека. Она сообщает, что функция была вызвана в скалярном контексте из строки 4 файла *Ambulation.pm*. Она показывает также, что функция была вызвана без каких-либо аргументов, т. е. как `&infested` вместо обычного `infested()`.
- Во второй строке показано, что функция `Ambulation::legs` была вызвана в списковом контексте из строки 5 файла *camel_flea* с четырьмя указанными аргументами.
- В третьей строке показано, что `main::pests` была вызвана в пустом контексте из строки 2 файла *camel_flea*.

Если имеются исполняемые команды фазы компиляции, такие как код в блоках `BEGIN` и `CHECK` или команды `use`, то обычно отладчик в них *не* останавливается, в отличие от `require` и блоков `INIT`, поскольку последние происходят после перехода к этапу выполнения (см. главу 18 «Компиляция»). Команды фазы компиляции можно трассировать, если установить опцию `AutoTrace` в `PERLDB_OPTS`.

Существуют некоторые возможности управления отладчиком Perl из самой программы Perl. Это можно сделать, например, для установки автоматической контрольной точки в некоторой подпрограмме, если данная программа выполняется под отладчиком. Однако из кода Perl можно передать управление отладчику, с помощью следующей команды, которая безвредна, если отладчик не запущен:

```
$DB::single = 1;
```

Если установить `$DB::single` равным 2, то получится эквивалент команды `n`, в то время как значение 1 эмулирует команду `s`. Чтобы моделировать команду `t`, нужно установить переменную `$DB::trace` равной 1.

Другим способом отладки модуля является установка точки останова на загрузку *load*:

```
DB<7> b load c:/perl/lib/Carp.pm
Will stop on load of `c:/perl/lib/Carp.pm'.
```

с последующим перезапуском отладчика командой `R`. Более точное управление можно осуществлять с помощью `b compile subname`, чтобы остановиться как можно скорее после компиляции заданной подпрограммы.

Команды отладчика

При вводе команд в отладчике не обязательно завершать их точкой с запятой. Используйте обратную косую черту для продолжения строк (но только при нахождении в отладчике).

Поскольку отладчик использует для выполнения команд `eval`, установки `my`, `our` и `local` уничтожаются после возврата из команды. Если команда отладчика совпадает по имени с какой-либо функцией в вашей собственной программе, предпошлите вызову функции какой-либо символ, отличный от команды отладчика, например `;` или `+`.

Если размер вывода встроенной команды отладчика превосходит количество строк на экране, укажите перед именем команды символ конвейеризации, и вывод будет произведен в постраничном формате:

```
DB<1> |h
```

У отладчика есть масса команд, которые мы разделяем (достаточно произвольно) на команды пошаговой обработки и прогона, работы с контрольными точками, трассировки, вывода, поиска кода, автоматического выполнения команд и, разумеется, «разные».

Вероятно, самой важной командой является `h`, выводящая подсказку. Если в ответ на приглашение отладчика ввести `h h`, то можно получить компактный листинг подсказки, умещающийся на одном экране. Если ввести `h` *COMMAND*, то вы получите подсказку по указанной команде отладчика.

Пошаговая обработка и прогон

Отладчик действует, проходя вашу программу строка за строкой. Следующие команды позволяют определить, что нужно проскочить и где нужно остановиться.

`s`

`s` *EXPR*

Команда отладчика `s` выполняет программу в пошаговом режиме. Это значит, что отладчик выполняет очередную строку программы, пока не достигнет следующей команды, при необходимости входя в подпрограмму. Если очередная строка содержит вызов функции, то отладчик останавливается на первой строке в этой функции. Если передается выражение *EXPR*, содержащее вызовы функций, последние также выполняются пошагово.

`n`

`n` *EXPR*

Команда `n` выполняет вызовы подпрограмм, не проходя их пошагово, до начала следующей команды на том же уровне (или более высоком). Если передается выражение *EXPR*, содержащее вызовы функций, эти функции будут выполняться с остановками перед каждой командой.

`<ENTER>`

Если просто нажать клавишу <Enter> в ответ на приглашение отладчика, будет повторно выполнена предыдущая команда `n` или `s`.

`.` Команда `.` возвращает внутренний указатель отладчика на последнюю выполненную строку и выводит эту строку.

r Эта команда продолжает выполнение до возврата из текущей выполняющейся подпрограммы. Она выводит возвращаемое значение, если установлен параметр PrintRet, что происходит по умолчанию.

Контрольные точки

b
b *LINE*
b *CONDITION*
b *LINE CONDITION*
b *SUBNAME*
b *SUBNAME CONDITION*
b postpone *SUBNAME*
b postpone *SUBNAME CONDITION*
b compile *SUBNAME*
b load *FILENAME*

Команда отладчика b устанавливает *контрольную точку* перед строкой *LINE*, приказывая отладчику остановить выполнение программы в этой точке и предоставляя вам возможность покопаться в данных. Если строка *LINE* опущена, контрольная точка устанавливается на строке, которая должна выполняться следующей. Если задано условие *CONDITION*, то оно проверяется каждый раз, когда должна выполниться команда: остановка происходит, только если значение *CONDITION* истинно. Контрольные точки можно установить только на начальных строках выполняемых команд. Заметьте, что в условиях не используется if:

```
b 237 $x > 30
b 237 ++$count237 < 11
b 33 /pattern/i
```

Формат b *SUBNAME* устанавливает контрольную точку (возможно, условную) перед первой строкой указанной подпрограммы. *SUBNAME* может быть переменной, содержащей ссылку в коде, в этом случае *CONDITION* не поддерживается.

Есть несколько способов установить контрольные точки в коде, который может быть еще даже не скомпилирован. Формат b устанавливает контрольную точку (возможно, условную) перед первой строкой *SUBNAME* после того, как она будет скомпилирована.

Формат b устанавливает контрольную точку на первой команде, которая должна быть выполнена после компиляции *SUBNAME*. Заметьте, что в отличие от формата postpone, эта команда находится вне рассматриваемой подпрограммы, поскольку подпрограмма еще не вызывалась, а только компилировалась.

Формат b load устанавливает контрольную точку на первой выполняемой строке файла. Аргумент *FILENAME* должен представлять полный путь к файлу, как в значениях %INC.

`d`

`d` *LINE*

Эта команда удаляет контрольную точку со строки *LINE*; если последняя опущена, удаляется контрольная точка со строки, которая должна быть выполнена следующей.

`D` Эта команда удаляет все контрольные точки.

`L` Эта команда выводит список всех контрольных точек и действий.

`c`

`c` *LINE*

Эта команда продолжает выполнение, дополнительно предоставляя возможность установить одноразовую контрольную точку в указанной строке *LINE*.

Трассировка

`T` Эта команда прослеживает стек вызовов.

`t`

`t` *EXPR*

Эта команда переключает режим трассировки, так что выводится каждая выполняющаяся строка программы. См. также обсуждение параметра `AutoTrace` далее в этой главе. Если задано выражение *EXPR*, отладчик трассирует его выполнение. См. также раздел «Автоматическое выполнение» далее в данной главе.

`W`

`W` *EXPR*

Эта команда добавляет *EXPR* к списку глобальных контрольных выражений (watch expressions). Контрольным называется выражение, вызывающее остановку при изменении его значения. Если *EXPR* не задано, все контрольные выражения удаляются.

Вывод данных

В отладчике Perl есть несколько команд, позволяющих изучать структуры данных, когда программа останавливается в контрольной точке.

`p`

`p` *EXPR*

Эта команда оказывает такое же действие, как `DB::OUT` *EXPR* в текущем пакете. В частности, поскольку это лишь собственная функция `print`, принадлежащая Perl, вложенные структуры данных и объекты не отображаются, для них следует использовать команду `x`. Обработчик `DB::OUT` осуществляет вывод на терминал (или в окно редактора) независимо от того, куда переадресован стандартный вывод.

x

x *EXPR*

Команда x вычисляет свое выражение в контексте списка и выводит результат в «красивом» виде. Это означает, что вложенные структуры выводятся рекурсивно при надлежащей перекодировке символов, не имеющих графического представления.

V

V *PKG*

V *PKG VARS*

Эта команда выводит все (или только некоторые, если заданы *VARS*) переменные в указанном пакете *PKG* (или, по умолчанию, пакете main), используя «красивую» печать. В хешах показываются ключи и значения, управляющие символы выводятся наглядно, встроенные структуры выводятся в удобном виде и т. д. Команда аналогична вызову команды x для всех нужных переменных, за исключением того, что x работает и с лексическими переменными. Кроме того, идентификаторы нужно задавать *без* спецификаторов типа, таких как $ или @, например:

```
V Pet::Camel SPOT FIDO
```

Вместо имен переменных в *VARS* можно использовать ~*PATTERN* или !*PATTERN*, чтобы вывести переменные, имена которых соответствуют (или не соответствуют) заданному шаблону.

X

X *VARS*

Эта команда представляет собой то же, что и V *CURRENTPACKAGE*, где *CURRENTPACKAGE* является пакетом, в который была скомпилирована текущая строка.

H

H -*NUMBER*

Эта команда выводит заданное количество *NUMBER* последних команд. В буфере запоминаются только команды длиннее одного символа (иначе большинство из них было бы s или n). Если аргумент *NUMBER* опущен, выводятся все команды.

Поиск кода

С помощью этих команд можно, находясь в отладчике, обнаруживать и выводить участки программы.

l

l *LINE*

l *SUBNAME*

l *MIN+INCR*

l *MIN-MAX*

Команда l выводит несколько следующих строк программы или указанную строку *LINE* либо несколько первых строк подпрограммы *SUBNAME*, либо указанную часть кода.

В формате *MIN+INCR* выводятся *INCR*+1 строк, начиная с *MIN*. В формате l *MIN-MAX* выводятся строки с номерами с *MIN* по *MAX*.

- Эта команда выводит несколько предыдущих строк программы.

w

w *LINE*

Выводит окно (несколько строк), окружающее заданную строку *LINE*, если она задана, или текущую строку, если строка *LINE* не указана.

f *FILENAME*

Эта команда позволяет посмотреть другую программу или команду eval. Если *FILENAME* не является полным именем пути, который можно найти в значениях %INC, то интерпретируется как регулярное выражение для нахождения нужного вам файла.

/*PATTERN*/

Эта команда осуществляет поиск в коде программы шаблона *PATTERN* в прямом направлении; замыкающий символ / не обязателен. Как и весь шаблон *PATTERN*: если он опущен, повторяется предыдущий поиск.

?*PATTERN*?

Эта команда осуществляет поиск шаблона *PATTERN* в коде программы в обратном направлении; замыкающий символ ? не обязателен. Если *PATTERN* опущен, повторяется предыдущий поиск.

S

S *PATTERN*

S !*PATTERN*

Команда S выводит имена подпрограмм, соответствующие (или, при задании !, не соответствующие) шаблону *PATTERN*. Если шаблон *PATTERN* не задан, перечисляются все подпрограммы.

Действия и выполнение команд

Находясь в отладчике, можно задать действия, которые должны быть выполнены в определенные моменты. Можно также запускать внешние программы.

a

a *COMMAND*

a *LINE*

a *LINE COMMAND*

Эта команда устанавливает действие, которое должно быть выполнено перед выполнением строки *LINE* или текущей строки, если номер строки

опущен. Например, следующая команда выводит `$foo` при каждом достижении строки 53:

```
a 53 print "DB FOUND $foo\n"
```

Если параметр `COMMAND` не задан, действие для заданной строки *LINE* удаляется. Если не заданы ни *LINE*, ни *COMMAND*, удаляется действие для текущей строки.

`A` Команда отладчика `A` удаляет все действия.

`<`

`< ?`

`< EXPR`

`<< EXPR`

В формате `< EXPR` задается выражение Perl, которое должно вычисляться перед каждым выводом приглашения отладчиком. Можно добавить еще одно выражение, используя формат `<< EXPR`, вывести список выражений с помощью `< ?` и удалить все выражения простым `<`.

`>`

`> ?`

`> EXPR`

`>> EXPR`

Команды `>` выполняются как родственные командам `<`, но выполняются после вывода приглашения отладчиком, а не перед ним.

`{`

`{ ?`

`{ COMMAND`

`{{ COMMAND`

Команды отладчика `{` выполняются аналогично `<`, но задают не выражение Perl, а команду отладчика, которую нужно выполнить перед выводом приглашения. Если вместо нее ошибочно введен блок кода, выдается предупредительное сообщение. Если это как раз то, что вам требуется, введите `;{ ... }` или даже `do { ... }`.

`!`

`! NUMBER`

`! -NUMBER`

`! PATTERN`

Одиночный `!` повторяет предыдущую команду. *NUMBER* определяет, какую команду из буфера нужно выполнить; например, `! 3` выполняет третью команду, введенную в отладчик. Если аргументу *NUMBER* предшествует символ «-», команды отсчитываются в обратном направлении: `! -3` выполняет третью от конца команду. Если вместо числа задается шаблон

PATTERN (без косых черт), выполняется последняя команда, начало которой совпадает с шаблоном. См. также параметр отладчика recallCommand.

!! *CMD*

Эта команда отладчика выполняет в подпроцессе внешнюю команду *CMD*, которая получит входные данные из DB::IN и запишет выходные в DB::OUT. См. также параметр отладчика shellBang. Эта команда использует оболочку, указанную в $ENV{SHELL}, что иногда может мешать правильной интерпретации статуса, сигнала и данным дампа памяти. Если нужно, чтобы команда возвращала при выходе правильное значение, установите переменную $ENV{SHELL} равной /bin/sh.

|

|*DBCMD*

||*PERLCMD*

Команда |*DBCMD* выполняет команду отладчика *DBCMD*, конвейеризуя DB::OUT на $ENV{PAGER}. Она часто используется с командами, которые выводят большой объем информации, например:

```
DB<1> |V main
```

Заметьте, что это относится к командам отладчика, а не к командам, вводимым из оболочки. Если вы хотите направить результат выполнения команды *who* на программу *more*, это можно сделать так:

```
DB<1> !!who | more
```

Команда ||*PERLCMD* аналогична |*DBCMD*, но, кроме того, временно выбирает DB::OUT, поэтому все команды, вызывающие print, printf или write без дескриптора файла, будут тоже отправляться по конвейеру. Например, если некая функция генерирует большой объем информации, выводимой посредством print, для постраничного вывода данных вместо предыдущей следует использовать такую команду:

```
DB<1> sub saywho { print "Users: ", `who` }
DB<2> ||saywho()
```

Разные команды

q *u* ^D

С помощью этих команд осуществляется выход из отладчика. Это предпочтительный способ выхода, хотя иногда действует дважды введенный exit. Установите параметр inhibit_exit равным 0, если хотите остаться в отладчике по окончании прогона программы. Может также потребоваться установить $DB::finished, если вы хотите проследить процесс уничтожения глобальных объектов в пошаговом режиме.

R Перезапустить отладчик, начав новый сеанс. Отладчик пытается сохранить преемственность работы между различными сеансами, однако неко-

торые установки и параметры командной строки могут быть утеряны. В настоящее время сохраняются следующие установки: буфер команд, контрольные точки, действия, параметры отладчика и параметры командной строки Perl `-w`, `-I` и `-e`.

`=`

`= ALIAS`

`= ALIAS VALUE`

Эта команда выводит текущее значение псевдонима *ALIAS*, если не задано значение *VALUE*. Если оно задано, то определяет новую команду отладчика с именем *ALIAS*. Если опущены как *ALIAS*, так и *VALUE*, выводится список всех текущих псевдонимов, например:

```
= quit q
```

ALIAS должен быть простым идентификатором и транслироваться также в простой идентификатор. Более сложные псевдонимы можно создавать путем добавления собственных записей непосредственно в `%DB::aliases`. См. раздел «Настройка отладчика» далее в этой главе.

`man`

`man MANPAGE`

Эта команда вызывает просмотр указанной страницы с помощью установленного в системе по умолчанию средства просмотра документации или само это средство, если *MANPAGE* опущен. Если этим средством просмотра является *man*, то для его вызова используются данные из текущего `%Config`. При необходимости автоматически добавляется префикс «`perl`»,что позволяет вводить в отладчике `man debug` и `man op`.

В тех системах, в которых обычно отсутствует утилита *man*, отладчик вызывает *perldoc*; для того чтобы изменить этот режим, установите в `$DB::doccmd` имя того средства просмотра, которое вам необходимо. Это можно сделать с помощью файла *rc* или непосредственным присвоением.

`O`

`O OPTION ...`

`O OPTION? ...`

`O OPTION=VALUE...`

Команда `O` позволяет манипулировать опциями отладчика, перечисленными в разделе «Опции отладчика» далее в этой главе. Формат команды `O OPTION` устанавливает каждую из перечисленных опций в 1. Если за *OPTION* следует вопросительный знак, выводится текущее значение опции.

В формате команды `O OPTION=VALUE` устанавливаются значения опций; если внутри *VALUE* есть пробельный символ, он должен быть заключен в кавычки. Например, можно задать `O pager="less -MQeicsNfr"`, чтобы *less* использовалась с этими конкретными флагами. Кавычки могут быть либо одинарными, либо двойными, но при этом нужно использовать управляю-

щие символы для ввода внутри строки того типа кавычек, в которые заключена строка. Необходимо также предварять управляющим символом обратную косую черту (обратный слэш), непосредственно предшествующую кавычке, но не предназначенную быть управляющим символом для ввода этой кавычки. Иными словами, выполняйте правила заключения в одинарные кавычки независимо от того, какой тип кавычек используется на самом деле. В ответ на команду установки опции отладчик выводит ее значение, используя при этом обозначения в одинарных кавычках:

```
DB<1> O OPTION='this isn\'t bad'
                OPTION = 'this isn\'t bad'
DB<2> O OPTION="She said, \"Isn't it?\""
                OPTION = 'She said, "Isn\'t it?"'
```

В силу исторических причин параметр =*VALUE* не обязателен, но значения, равные единице, по умолчанию устанавливаются только там, где это допустимо, т. е. по большей части для логических опций. Лучше назначать конкретные значения *VALUE* с помощью =. *OPTION* можно сокращать, но делать это, пожалуй, не следует, если только загадочность и непонятность не являются вашей целью. Одновременно можно установить несколько опций. Их список приведен в разделе «Опции отладчика».

Настройка отладчика

Есть достаточно много способов конфигурирования отладчика, поэтому вам вряд ли потребуется модифицировать его. Находясь в самом отладчике, можно изменить его режим с помощью команды `O`, из командной строки это можно сделать через переменную окружения `PERLDB_OPTS` и путем выполнения предварительно заданных команд, хранимых в файлах *rc*.

Поддержка отладки в редакторах

Механизм буфера команд отладчика не позволяет производить редактирование в командной строке, как это делают многие оболочки: нельзя извлекать предыдущие строки с помощью `^p` или перемещаться к началу строки с помощью `^a`, хотя можно выполнять предыдущие строки, используя синтаксис с восклицательным знаком, знакомый пользователям оболочки. Однако если установить модули `Term::ReadKey` и `Term::ReadLine`, имеющиеся в CPAN, можно получить полные возможности редактирования, аналогичные имеющимся в GNU *readline* (3).

Если у вас установлен *emacs*, то можно организовать его взаимодействие с отладчиком Perl для создания интегрированной среды разработки программного обеспечения, напоминающей взаимодействие *emacs* с отладчиками C. Вместе с Perl поставляется выполняемый файл, позволяющий модифицировать *emacs* так, чтобы он работал как синтаксически управляемый редактор, понимающий (частично) синтаксис Perl. Посмотрите в каталог *emacs* дистрибутива исходного кода Perl. Пользователям *vi* следует также

присмотреться к *vim* (и его мышино-оконной свободно распространяемой версии *gvim*), позволяющему выделять цветом ключевые слова Perl.

Есть также разработанная одним из нас (Томом) аналогичная по возможностям надстройка для взаимодействия с любым поставляемым *vi* и оконной системой X11. Она действует аналогично интегрированной многооконной среде, предоставляемой *emacs*, в которой отладчик управляет редактором. На момент написания данной книги оставалось неясным, войдет ли она в итоге в дистрибутив Perl, но мы решили сообщить вам о такой возможности.

Настройка с помощью файлов инициализации

Некоторые настройки можно произвести, внеся изменения в файлы *.perldb* или *perldb.ini* (в зависимости от вашей операционной системы), в которых содержится код инициализации. В этом файле инициализации содержится код Perl, а не команды отладчика, и он обрабатывается раньше, чем анализируется переменная окружения `PERLDB_OPTS`. Например, можно следующим образом создавать псевдонимы, добавляя элементы в хеш `%DB::alias`:

```
$alias{len}  = 's/^len(.*)/p length($1)/';
$alias{stop} = 's/^stop (at|in)/b/';
$alias{ps}   = 's/^ps\b/p scalar /';
$alias{quit} = 's/^quit(\s*)/exit/';
$alias{help} = 's/^help\s*$/|h/';
```

Можно изменить опции, используя в файле инициализации вызовы функций внутреннего API отладчика:

```
parse_options("NonStop=1 LineInfo=db.out AutoTrace=1 frame=2");
```

Если в файле инициализации определена подпрограмма `afterinit`, то по окончании инициализации отладчика будет вызвана эта подпрограмма. Файл инициализации может находиться в текущем каталоге или исходном каталоге. Поскольку файл может содержать произвольные команды Perl, то по соображениям безопасности его владельцем должен быть суперпользователь (root) или текущий пользователь, а запись в него должна быть разрешена только его владельцу.

Если нужно модифицировать отладчик, скопируйте *perl5db.pl* из библиотеки Perl под новым именем и делайте с ним все, что вашей душе угодно. Затем вам потребуется установить переменную окружения `PERL5DB` примерно таким образом:

```
BEGIN { require "myperl5db.pl" }
```

В качестве последнего варианта можно использовать для настройки отладчика переменную `PERL5DB`, непосредственно устанавливая внутренние переменные или вызывая внутренние функции отладчика. Учтите, однако, что все переменные и функции, не документированные здесь или в элект-

ронных страницах руководства *perldebug*, *perldebguts* или *DB*, предназначены только для внутреннего употребления и могут быть изменены без предупреждения.

Опции отладчика

Отладчик имеет многочисленные опции, которые можно установить с помощью команды `O` – интерактивно, через среду или файл инициализации.

`recallCommand, ShellBang`

Символы, используемые для повторного вызова команды или вызова оболочки. По умолчанию тот и другой устанавливаются равными `!`.

`pager`

Программа, работающая как устройство вывода команд, конвейеризуемых на пейджер – программу постраничного вывода (начинающихся с символа "`|`"). По умолчанию используется `$ENV{PAGER}`. Поскольку отладчик руководствуется текущими настройками терминала для полужирного шрифта и подчеркивания, то если выбранная программа постраничного вывода не пропускает escape-последовательности в неизменном виде, некоторые команды отладчика при прохождении через пейджер оказываются нечитаемыми.

`tkRunning`

Запуск под модулем `Tk` при выводе подсказки (с `ReadLine`).

`signalLevel, warnLevel, dieLevel`

Установить уровень детализации сообщений. По умолчанию отладчик не трогает ваши обработки исключительных ситуаций и предупредительные сообщения, поскольку их изменение может нарушить работу правильно выполняющихся программ.

Чтобы отключить этот безопасный режим, установленный по умолчанию, задайте этим величинам значения, большие 0. На уровне 1 вы сможете отслеживать получение любых предупредительных сообщений (это часто раздражает) или исключительных ситуаций (это часто ценно). К несчастью, отладчик не может различать фатальные и не фатальные исключительные ситуации. Если `dieLevel` равен 1, то нефатальные исключительные ситуации тоже трассируются и бесцеремонно изменяются, если наступают при выполнении `eval` для ваших строк или в модулях, которые вы пытаетесь загрузить. Если `dieLevel` равен 2, то отладчик не интересуется тем, откуда возникают исключительные ситуации: он захватывает права вашего обработчика исключительной ситуации и выводит трассировку, а затем модифицирует все исключительные ситуации, украшая их собственными деталями. Это, возможно, и полезно в некоторых задачах трассировки, но, скорее всего, безнадежно запутает любую программу, в которой обработка исключительных ситуаций организована достаточно серьезным образом.

Отладчик постарается вывести сообщение при поступлении неперехваченных сигналов INT, BUS или SEGV. Если вы находитесь в медленном системном вызове (таком как wait, accept или read с клавиатуры или сокета) и не установили собственный обработчик сигналов $SIG{INT}, то не сможете вернуться в отладчик с помощью <Ctrl>+<C>, поскольку обработчик сигнала $SIG{INT}, имеющийся в отладчике, не поймет, что должен возбудить исключительную ситуацию, чтобы выполнить *longjmp*(3) из медленного системного вызова.

AutoTrace

Установка режима трассировки (аналогична команде t, но может быть помещена в PERLDB_OPTS).

LineInfo

Назначение файла или конвейера, в который должны выводиться данные о номерах строк. Если это конвейер (например, |visual_perl_db), то используется короткое сообщение. Этот механизм применяется для взаимодействия с подчиненным редактором или визуальным отладчиком, таким как специальные обработчики (hooks) для *vi* или *emacs* или графический отладчик *ddd*.

inhibit_exit

Если значение равно 0, разрешен выход по окончании отладки программы.

PrintRet

Если опция установлена (по умолчанию – да), то выводится значение, возвращаемое командой r.

ornaments

Воздействует на характер вывода командной строки (см. электронную документацию по Term::ReadLine). В настоящее время нет способа отключить украшения (ornaments), из-за чего вывод на некоторых мониторах или с некоторыми пейджерами бывает нечитаемым. Считается дефектом программы.

frame

Управляет выводом сообщений при входе в подпрограммы и выходе из них. Если ложно frame & 2, то сообщения выводятся только при входе. (Вывод при выходе может быть полезен, если он чередуется с другими сообщениями.)

Если frame & 4 истинно, то выводятся аргументы функций вместе с данными о контексте и вызвавшей программе. Если истинно frame & 8, то включаются перегруженный stringify и tied FETCH для выводимых аргументов. Если истинно & 16, то выводится значение, возвращаемое подпрограммой.

Длина, по которой обрезается список аргументов, управляется следующей опцией.

maxTraceLen

Длина, по которой обрезается список аргументов при установленном бите 4 опции frame.

Следующие опции управляют поведением команд V, X и x:

arrayDepth, hashDepth

Выводить только первые *n* элементов. Если *n* опущено, выводятся все элементы.

compactDump, veryCompact

Изменить стиль вывода массивов и хешей. Если включена опция compactDump, короткие массивы могут выводиться на одной строке.

globPrint

Вывод содержимого таблицы типов данных typeglobs.

DumpDBFiles

Вывод массивов, содержащих отлаживаемые файлы.

DumpPackages

Вывод таблиц символов пакетов.

DumpReused

Вывод содержимого «повторно используемых» адресов.

quote, HighBit, undefPrint

Изменение стиля вывода строк. Значением по умолчанию для quote является auto; можно установить формат с двойными или одинарными кавычками, установив его в " или ', соответственно. По умолчанию символы с установленным старшим битом выводятся дословно.

UsageOnly

Когда включена эта опция, вместо содержимого переменных пакета выводятся элементарные дампы расходования памяти для каждого пакета на основании суммарного размера строк, обнаруженных в переменных пакета. Поскольку используется таблица символов пакета, лексические переменные игнорируются.

Автоматическое выполнение

В переменной $ENV{PERLDB_OPTS} можно поместить опции инициализации TTY, noTTY, ReadLine и NonStop. При начальном запуске опции инициализируются значениями из этой переменной.

Если в файле инициализации содержится строка:

```
parse_options("NonStop=1 LineInfo=tperl.out AutoTrace");
```

то программа работает без вмешательства оператора, а данные трассировки помещаются в файл *db.out*. (Если вы будете ее прерывать, то следует задать для `LineInfo` значение */dev/tty*, если хотите что-то увидеть.)

Следующие опции можно задать только при запуске. Чтобы установить их в файле инициализации, вызовите `parse_options("OPT=VAL")`.

`TTY`

Терминал, используемый для ввода-вывода при отладке.

`noTTY`

Если эта опция установлена, отладчик входит в режим `NonStop`. При прерывании (или явной передаче управления отладчику через установку `$DB::signal` или `$DB::single` в программе Perl) отладчик соединяется с терминалом, указанным при запуске в опции `TTY` или выбранным вами из найденных во время выполнения с помощью модуля `Term::Rendezvous`.

В этом модуле должен быть реализован метод с именем `new`, возвращающий объект с двумя методами: `IN` и `OUT`. Они должны возвращать дескрипторы файлов для использования отладчиком при вводе и выводе соответственно. Метод `new` должен проверить аргумент, содержащий значение `$ENV{PERLDB_NOTTY}` при запуске, либо `"/tmp/perldbtty$$"`. Этот файл не проверяется на правильность владения им или защищенность от записи, поэтому теоретически возможна угроза защите данных.

`ReadLine`

Если имеет значение «ложь», то отключается поддержка отладчиком `ReadLine`, для того чтобы отлаживать приложения, которые сами используют модуль `ReadLine`.

`NonStop`

Если эта опция установлена, то отладчик переходит в неинтерактивный режим до того, пока он не будет прерван или ваша программа не установит `$DB::signal` или `$DB::single`.

Иногда опции можно однозначно определить, обозначив их лишь первой буквой, но мы советуем всегда указывать их полностью для лучшей читаемости и обеспечения совместимости в будущем.

Вот пример использования переменной окружения `PERLDB_OPTS` для автоматической установки опций.[1] В нем программа запускается в неинтерактивном режиме с выводом информации о каждом входе в подпрограмму и для каждой выполняемой строки. Выходные данные трассировки помещаются в файл *tperl.out*. Это позволяет программе использовать обычные устройства ввода-вывода и избежать смешивания своих данных с данными трассировки.

```
$ PERLDB_OPTS="NonStop frame=1 AutoTrace LineInfo=tperl.out" perl -d myprog
```

[1] Мы применяем синтаксис оболочки *sh* для отображения установок переменных окружения. Пользователи других оболочек должны произвести соответствующие изменения.

Прервав выполнение программы, нужно скорее установить `O LineInfo=/dev/tty` или другое допустимое для вашей платформы устройство, иначе вы не увидите приглашения отладчика.

Поддержка отладчика

Perl предоставляет специальные функции для отладки как на этапе компиляции, так и на этапе выполнения, чтобы создавать такую же среду отладки, как для стандартного отладчика. Эти функции не следует путать с опциями *perl -D*, которые можно использовать, только если вы скомпилировали Perl с опцией `-DDEBUGGING`.

Например, при каждом обращении к встроенной функции Perl `caller` из пакета `DB` аргументы, которые были помещены в соответствующий стековый фрейм, копируются в массив `@DB::args`. При вызове Perl с ключом `-d` включаются следующие дополнительные функции:

- Perl вставляет перед первой строкой вашей программы значение `$ENV{PERL5DB}` (или `BEGIN {require 'perl5db.pl'}`, если ее нет).
- В массив `@{"_<$filename"}` помещаются строки с `$filename` для всех файлов, скомпилированных Perl. То же касается строк, выполняемых через `eval`, в которых содержатся подпрограммы, или выполняемых в данное время. `$filename` для строк в `eval` имеют вид `(eval 34)`. Утверждения кода в регулярных выражениях имеют вид `(re_eval 19)`.
- В хеш `%{"_<$filename"}` помещаются контрольные точки и действия, ключом для которых является номер строки. Отдельные элементы можно устанавливать противоположными всему хешу. Perl интересует здесь только булева истинность, хотя значения, используемые *perl5db.pl*, имеют вид `"$break_condition\0$action"`. Значения в этом хеше имеют чудесное (magic) свойство в числовом контексте: они равны нулю, если на строке нельзя установить контрольную точку. То же касается вычисляемых строк, в которых содержатся подпрограммы или выполняемых в данное время. `$filename` для строк, выполняемых через `eval`, имеет вид `(eval 34)` или `(re_eval 19)`.
- В скаляре `${"_<$filename"}` содержится `"_<$filename"`. Это имеет также место для вычисляемых строк, содержащих подпрограммы или выполняемых в данное время. `$filename` для строк, выполняемых через `eval`, имеет вид `(eval 34)` или `(re_eval 19)`.
- После компиляции каждого файла, загружаемого через `require`, но до его выполнения, вызывается `DB::postponed(*{"_<$filename"})`, если существует подпрограмма `DB::postponed`. В данном случае `$filename` является расширенным именем файла в соответствии со значениями в `%INC`.
- После компиляции всех подпрограмм *subname* проверяется `$DB::postponed{`*subname*`}`. Если такой ключ есть и если подпрограмма `DB::postponed` тоже существует, осуществляется вызов `DB::postponed(`*subname*`)`.

- Поддерживается хеш `%DB::sub`, ключами которого являются имена подпрограмм, а значения имеют вид *filename*: *startline-endline*. *filename* имеет вид `(eval 34)` для подпрограмм, определенных внутри `eval`, или `(re_eval 19)` для тех, которые находятся в утверждениях кода регулярных выражений.
- Когда выполнение программы доходит до места, в котором может содержаться контрольная точка, вызывается подпрограмма `DB::DB()`, если одна из переменных — `$DB::trace`, `$DB::single` или `$DB::signal` — имеет истинное значение. Эти переменные не локализуемы. Эта функция выключается, когда выполнение происходит внутри `DB::DB()`, в том числе для вызываемых из нее функций, если только не выполняется условие `$^D & (1<<30)`.
- Когда выполнение программы приводит к вызову подпрограммы, вместо него производится вызов `&DB::sub(`*args*`)`, при этом `$DB::sub` содержит имя вызываемой подпрограммы. Этого не происходит, если подпрограмма компилировалась в пакете `DB`.

Обратите внимание: если для работы `&DB::sub` требуются внешние данные, то до завершения работы невозможен вызов никаких подпрограмм. Для стандартного отладчика пример такой зависимости дает переменная `$DB::deep` (на какое количество уровней рекурсии можно пройти вглубь отладчика до обязательного выхода).

Создание собственного отладчика

Минимальный работающий отладчик состоит из одной строки:

```
sub DB::DB {}
```

которую, хотя она и абсолютно ничего не делает, легко можно определить через переменную окружения `PERL5DB`:

```
% PERL5DB="sub DB::DB {}" perl -d your-program
```

Другой компактный отладчик, несколько более полезный, можно создать так:

```
sub DB::DB {print ++$i; scalar <STDIN>}
```

Этот небольшой отладчик выводит последовательный номер каждой встретившейся команды и ждет нажатия перевода строки для продолжения работы.

Следующий отладчик, как ни мал он с виду, вполне функционален:

```
{
    package DB;
    sub DB  {}
    sub sub {print ++$i, " $sub\n"; &$sub}
}
```

Он выводит порядковый номер вызова подпрограммы и имя вызываемой подпрограммы. Заметьте, что `&DB::sub` должна компилироваться из пакета `DB`, как мы здесь это сделали.

Если базой для вашего нового отладчика является текущий отладчик, существует несколько обработчиков, которые вам помогут его настроить. При запуске отладчик читает файл начальной конфигурации из текущего каталога или исходного каталога. После считывания файла отладчик читает переменную окружения `PERLDB_OPTS` и выполняет ее анализ как остатка строки `O ...`, которая могла быть введена в ответ на приглашение отладчика.

Отладчик поддерживает также «магические» внутренние переменные, такие как `@DB::dbline`, `%DB::dbline`, служащие псевдонимами для `@{&quot:::_<current_file"} %{":: _<current_file"}`. Здесь *current_file* является файлом, выбранным в данный момент либо явно с помощью команды отладчика `f`, либо неявно в соответствии с логикой выполнения программы.

Настройку облегчают некоторые функции. `DB::parse_options(STRING)` выполняет анализ строки как опции команды `O`. `DB::dump_trace(SKIP [,COUNT])` пропускает заданное количество фреймов и возвращает список, содержащий сведения о стеках вызова или обо всех них, если отсутствует *COUNT*. Каждый элемент его является ссылкой на хеш с ключами `"context"` (`.`, `$` или `@`), `"sub"` (имя подпрограммы или сведения о `eval`), `"args"` (`undef` или ссылка на массив), `"file"` и `"line"`. `DB::print_trace(FH, SKIP [,COUNT [,SHORT]])` выводит отформатированные сведения о стеках вызова по заданному дескриптору файла. Две последние функции удобно использовать как аргументы команд отладчика `"<"` и `"<<"`.

Нет необходимости все это учить – большинству из нас. На деле, когда требуется отладить программу, мы обычно добавляем в код некоторое количество команд вывода и повторно прогоняем программу.

Хорошо еще, если мы вспомним, что надо бы включить вывод предупреждающих сообщений. Этого часто оказывается достаточно, чтобы обнаружить проблему и не рвать на голове волосы (или то, что от них осталось). Однако если этого оказывается недостаточно, приятно сознавать, что за ключом `-d` в терпеливом ожидании находится чудный отладчик, который может сделать чуть ли не все, *за исключением* того, чтобы найти за вас ошибку.

Самое важное, что следует запомнить о настройке отладчика, это, пожалуй, следующее: не ограничивайте свое представление о программных ошибках (bugs) как о вещах, которые приводят в расстройство Perl. Программная ошибка существует и тогда, когда программа приводит в расстройство *вас*. Выше мы показали пару очень простых пользовательских отладчиков. В следующем разделе мы рассмотрим пользовательский отладчик иного сорта, который в состоянии (иногда) помочь отладить ошибку под названием «Эта штука собирается когда-нибудь закончить работу?».

Профайлер Perl

Хотите, чтобы ваша программа работала быстрее? Ну, конечно, да. Но прежде следует спросить себя: «Действительно ли нужно тратить время на то, чтобы эта программа работала быстрее?» Занимательная оптимизация может быть развлечением,[1] но обычно есть лучшие способы потратить время. Иногда можно заранее составить план и запустить программу, а самому пойти выпить чашечку кофе. (Или использовать это обстоятельство в качестве объяснения своей отлучки.) Но если действительно требуется, чтобы ваша программа работала быстрее, начать следует с ее профилирования. Профайлер может сообщить, какие участки программы дольше всего выполняются, в результате вы не будете терять время, совершенствуя подпрограмму, оказывающую незначительное влияние на общее время выполнения.

Вместе с Perl поставляется и профайлер, модуль `Devel::DProf`. Его можно использовать для профилирования программы Perl в *mycode.pl*, введя:

```
perl -d:DProf mycode.pl
```

Хотя мы и назвали `DProf` профайлером, поскольку именно в этом состоит его функция, в нем используется тот же самый механизм, который мы обсуждали ранее в этой главе. `DProf` является просто отладчиком, регистрирующим моменты времени, в которые Perl вошел или вышел из каждой подпрограммы.

Когда профилируемый сценарий завершается, `DProf` сохраняет дамп информации о времени в файл с именем *tmon.out*. Программа *dprofpp*, поставляемая с Perl, умеет анализировать *tmon.out* и создавать отчет. Можно использовать *dprofpp* и как внешний интерфейс для процесса в целом с помощью ключа `-p`, описываемого далее.

Пусть есть такая программа:

```
outer();

sub outer {
  for (my $i=0; $i < 100; $i++) { inner() }
}

sub inner {
  my $total = 0;
  for (my $i=0; $i < 1000; $i++) { $total += $i }
}

inner();
```

dprofpp может вывести следующую информацию:

```
Total Elapsed Time = 0.537654 Seconds
  User+System Time = 0.317552 Seconds
```

[1] По крайней мере, так считает Натан Торкингтон, который написал этот раздел книги.

```
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 85.0   0.270  0.269    101   0.0027 0.0027  main::inner
 2.83   0.009  0.279      1   0.0094 0.2788  main::outer
```

Обратите внимание на то, что сумма процентов не составляет 100. На самом деле, в нашем случае она довольно далека от этого значения, и это должно послужить указанием на то, что программу нужно прогонять дольше. Общее правило заключается в том, что чем больше данных профилирования можно получить, тем лучше статистическая выборка. Если увеличить число выполнения внешнего цикла со 100 до 1000, то получатся более точные результаты:

```
Total Elapsed Time = 2.875946 Seconds
  User+System Time = 2.855946 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 99.3   2.838  2.834   1001   0.0028 0.0028  main::inner
 0.14   0.004  2.828      1   0.0040 2.8280  main::outer
```

В первой строке сообщается о том, сколько времени выполнялась программа, от начала и до конца. Во второй строке выводятся два числа: время, потраченное на выполнение пользовательского кода («user»), и время, потраченное операционной системой на выполнение системных вызовов, осуществляемых из вашей программы («system»). (Не будем обращать внимания на ложную точность представления этих чисел: генератор тактовых импульсов компьютера наверняка не отмечает миллионные доли секунды, в лучшем случае это сотые.)

Время «user+system» можно изменить с помощью параметров командной строки *dprofpp*. `-r` осуществляет вывод истекшего времени, `-s` – только системного и `-u` – только времени пользователя.

Остальная часть отчета содержит распределение времени по подпрограммам. Строка «Exclusive Times» означает, что когда подпрограмма `outer` вызывала подпрограмму `inner`, время, затраченное в `inner`, не включалось во время, потраченное `outer`. Чтобы изменить этот режим и складывать время `inner` с временем `outer`, задайте для *dprofpp* параметр `-I<`.

Для каждой подпрограммы сообщается следующее: `%Time` – процент времени, потраченного при вызове этой подпрограммы; `ExclSec` – время в секундах, потраченное в этой подпрограмме, исключая время работы вызванных из нее подпрограмм; `CumulS` – время в секундах, потраченное в этой подпрограмме и вызванных из нее подпрограммах; `#Calls` – число вызовов подпрограммы; `sec/call` – среднее время в секундах каждого вызова подпрограммы, не включая время работы подпрограмм, вызванных из нее; `Csec/c` – среднее время в секундах каждого вызова подпрограммы вместе с работой подпрограмм, вызванных из нее.

Наиболее полезной из этих характеристик является `%Time`, которая показывает, куда уходит время. В нашем случае большую часть времени занимает процедура `inner`, поэтому следует попытаться оптимизировать ее или придумать алгоритм, при котором она вызывалась (использовалась) бы реже. `:-)`

Опции *dprofpp* предоставляют доступ к другим данным или изменяют способ подсчета времени. Можно также заставить *dprofpp* сначала запустить сценарий, поэтому помнить ключ `-d:Dprof` не обязательно:

`-p` *SCRIPT*

Сообщает *dprofpp* о необходимости профилировать указанный сценарий *SCRIPT*, а затем интерпретировать данные профилирования. См. также `-Q`.

`-Q` Используется вместе с `-p` для сообщения *dprofpp* о необходимости завершить работу после выполнения профилирования без интерпретации данных.

`-a` Сортировать выводимую информацию в алфавитном порядке по именам подпрограмм, а не по убыванию доли времени.

`-R` Вести отдельный учет по анонимным процедурам, определенным в том же пакете. По умолчанию все анонимные процедуры учитываются как одна с именем `main::__ANON__`.

`-I` Вывести время работы всех подпрограмм с учетом времени работы дочерних подпрограмм.

`-l` Сортировать по количеству обращений к подпрограмме. Может помочь определить кандидатов на подстановку (inlining).

`-O` *COUNT*

Показать только первые *COUNT* подпрограмм. По умолчанию равно 15.

`-q` Не выводить заголовки колонок.

`-T` Вывести на стандартное устройство вывода дерево вызова подпрограмм. Статистические данные по подпрограммам не выводятся.

`-t` Вывести на стандартное устройство вывода дерево вызова подпрограмм. Статистические данные по подпрограммам не выводятся. Функция, вызванная несколько раз (последовательно) на одном уровне вызова выводится один раз с указанием счетчика обращений.

`-S` Структурировать выдачу согласно порядку вызова подпрограмм:

```
main::inner x 1         0.008s
main::outer x 1         0.467s = (0.000 + 0.468)s
   main::inner x 100     0.468s
```

Это нужно понимать так: с верхнего уровня программы `inner` была вызвана один раз и работала 0,008 с, с верхнего уровня программы `outer` была вызвана один раз и работала 0,467 с, включая вызываемые подпрограммы (0 с в самой `outer` и 0,468 с в подпрограммах, вызванных из `outer`) и вызвав `inner` (которая работала 0,468 с) 100 раз. Ну как, все понятно?

Ветви от одного и того же уровня (например, один раз вызвана `inner` и один раз вызвана `outer`) сортируются по суммарному времени.

`-U` Не сортировать. Выводить согласно порядку обнаружения в исходных данных профилирования.

`-v` Сортировать по среднему времени нахождения в подпрограмме для всех вызовов. Иногда помогает выявить кандидатов на оптимизацию вручную путем подстановки (inlining) тел подпрограмм.

`-g subroutine`

Игнорировать подпрограммы, исключая *subroutine* и вызовы из нее.

Прочие параметры описаны в *dprofpp*(1), стандартной странице руководства по этой программе.

`DProf` — не единственная возможность для осуществления профилирования. В CPAN есть также программа `Devel::SmallProf`, сообщающая время, потраченное в каждой строке программы. Она поможет выявить использование какой-либо конструкции Perl, которая оказывается исключительно дорогостоящей. Большинство встроенных функций достаточно эффективны, но легко случайно написать регулярное выражение, накладные расходы на применение которого возрастают экспоненциально в зависимости от объема входных данных. Другие полезные советы можно найти в разделе «Эффективность» главы 24 «Распространенные приемы программирования».

Теперь можно сделать перерыв и выпить тот самый кофе. Для чтения следующей главы это необходимо.

21

Внутри и снаружи

Как уже говорилось в главе 18 «Компиляция», *perl* (программа) содержит в себе как компилятор, так и интерпретатор программ, написанных на Perl (языке). Сам компилятор/интерпретатор Perl написан на C. В этой главе мы в общих чертах опишем, как работает эта C-программа с точки зрения тех, кто хотел бы дополнить Perl или встроить его в свою программу. Когда *расширяют* Perl, фрагмент C-кода, называемый *расширением* (*extension*) передается под управление Perl, а при *встраивании* (*embedding*) Perl интерпретатор Perl[1] передается под управление более крупной C-программы.

Краткое описание, которое мы здесь приводим, не может заменить электронную документацию по внутреннему устройству Perl: см. документацию по *perlguts*, *perlxs*, *perlxstut*, *perlcall*, *perlapi* и *h2xs*, поставляемым с Perl. С другой стороны, тем, кто не собирается дополнять или встраивать Perl, знание этого материала никогда не понадобится.

Те же, кому этот материал нужен, сначала должны получить некоторое представление о внутреннем устройстве Perl. Для понимания последующего материала по большей части необходимо также знание C. Для запуска примеров

---

[1] Хотя мы стараемся отличать компилятор от интерпретатора в тех случаях, когда это различие важно, все время повторять «компилятор/интерпретатор» становится несколько утомительным, поэтому мы часто для краткости говорим «интерпретатор», чтобы обозначить весь массив кода C и данных, действующий как один экземпляр *perl* (программы); при встраивании Perl можно иметь несколько экземпляров интерпретатора, каждый из которых ведет себя как самостоятельный маленький *perl*.

понадобится компилятор C. Если ваша конечная цель состоит в создании модулей, которыми смогут пользоваться другие люди, им тоже понадобится компилятор C. Многие из приведенных примеров смогут работать только в Unix-подобных системах. И кроме того, этот материал может подвергнуться изменениям в последующих версиях Perl.

Другими словами, мы на границе страны драконов.

Как работает Perl

Когда на вход компилятора Perl поступает программа Perl, то в первую очередь он приступает к *лексическому анализу*: разбивает программу на основные синтаксические элементы, часто называемые *лексемами* (*tokens*). Если взять программу:

```
print "Hello, world!\n";
```

то лексический анализатор выделит в ней три лексемы: `print`, `"Hello, world!\n"` и завершающий символ точки с запятой. После этого проводится *синтаксический разбор* (*parsing*) последовательности лексем, устанавливающий взаимосвязь между лексемами. В Perl граница между лексическим и синтаксическим анализом более расплывчата, чем в других языках. (Имеется в виду в других компьютерных языках. Вы поймете, с чем это связано, если представите себе, какой различный смысл может иметь `new Critter` в зависимости от того, существуют ли пакет Critter или подпрограмма с именем `new`. С другой стороны, мы постоянно разрешаем такого рода неопределенности, разговаривая на естественных языках.)

После проведения синтаксического анализа программы и, предположительно, ее осмысления программа компилируется в дерево *кодов операций* (*opcodes*), представляющих операции низкого уровня, и, наконец, выполняется это дерево операций – если только Perl не был вызван с ключом `-c` («проверка синтаксиса»), вызывающим выход по завершении этапа компиляции. Именно на этапе компиляции, а не исполнения, выполняются блоки `BEGIN`, `CHECK` и команды `use`.

Внутренние типы данных

Когда обрабатывается дерево кодов операций, составляющее скомпилированную программу Perl, данные Perl создаются, обрабатываются и уничтожаются. Для всех известных типов данных Perl есть соответствующие типы данных C, используемые внутри Perl, о которых нужно знать при передаче данных между двумя языками.

Три определения типа (typedef) в C соответствуют основным типам данных Perl: `SV` (скалярное значение), `AV` (значение типа массив) и `HV` (значение типа хеш). Кроме того, `IV` представляет простой тип целого со знаком, по размеру достаточного для хранения указателя или целого числа, а `I32` и `I16` являются

типами, которые гарантированно могут содержать 32 и 16 бит соответственно. Для хранения беззнаковых разновидностей последних трех типов существуют отдельные типы UV, U32 и U16. Все эти типы данных могут обрабатываться функциями C, описанными в документации *perlguts*. Ниже мы кратко опишем принцип работы некоторых из этих функций:

- Есть четыре типа данных, которые можно поместить в SV: целое число (IV), действительное число с двойной точностью (NV), строка (PV) и другой скаляр (SV). Для SV существуют десятки функций, позволяющих создавать, изменять, наращивать, проверять истинность и определенность представляемых ими скаляров Perl. Ссылки Perl реализованы как RV, особый тип SV.
- При создании AV массив пустой или в нем размещаются SV, что оправданно, поскольку массив представляет собой совокупность скаляров.
- С HV связаны C-функции для записи, выборки, удаления и проверки существования пар ключ/значение в хеше, представляемом HV.
- Есть также тип GV (значение glob), в котором может храниться ссылка на любое значение, связанное с идентификатором переменной: скалярную величину, массив, хеш, подпрограмму, дескриптор ввода/вывода или формат.

При расширении языка Perl требуется знать об этих величинах для связывания с функциями C. При встраивании Perl об этих величинах нужно знать для обмена данными с интерпретатором Perl, включаемым в программу на C.

Расширение Perl (использование C из Perl)

При желании использовать исходный код C или библиотеку C из Perl необходимо создать библиотеку, которая может динамически загружаться исполняемым модулем *perl* или связываться с ним статически. (Обычно предпочитают динамическую загрузку, чтобы минимизировать число исполняемых модулей *perl*.) Эта библиотека создается с помощью *XS*-файла (с расширением *.xs*), содержащего ряд подпрограмм-оболочек. Эти подпрограммы-оболочки не являются подпрограммами Perl, а написаны на языке XS и носят название *XSUB* (от «eXternal SUBroutine» – «внешняя подпрограмма»). XSUB может служить оболочкой для функции C из внешней библиотеки, функции C, находящейся в другом файле XS, или чистого кода C в самой XSUB. После этого с помощью утилиты *xsubpp*, поставляемой с Perl, файл XS транслируется в C-код, который можно скомпилировать в библиотеку, понятную Perl.

Если операционная система поддерживает динамическую компоновку, то в итоге мы получим модуль Perl, работа с которым происходит так же, как с любым другим модулем, на 100% написанным на чистом Perl, хотя внутренне он использует откомпилированный код C. Этот модуль извлекает аргументы из стека аргументов Perl, преобразует величины Perl в формат, необходимый для конкретной функции C (указанной в объявлении XSUB), вызывает функцию C и, в конечном счете, передает значения, возвращенные

функцией, обратно в Perl. Возвращаемые значения могут передаваться Perl путем размещения их в стеке Perl или изменения аргументов, которые были переданы из Perl. (Если система не поддерживает динамическую компоновку, то предстоит дополнительное испытание, о чем мы расскажем в следующем разделе.)

Приведенное описание несколько упрощенно изображает то, что происходит на самом деле. Поскольку соглашения по вызову в Perl более гибкие, чем в C, подпрограммы XSUB на практике могут осуществлять многие дополнительные действия, такие как проверка допустимости входных параметров, возбуждение исключительных ситуаций, возврат `undef` или пустого списка, вызов разных функций C в зависимости от числа и типа аргументов или предоставление объектно-ориентированного интерфейса. Еще раз советуем посмотреть страницы электронного руководства *perlxs* и *perlxstut*.

Язык XS и подпрограммы XSUB

XS предоставляет просто удобство: ничто не мешает нам написать связующий код непосредственно на C и скомпоновать его с исполняемым модулем Perl. Однако это утомительно, особенно если нужно писать связующий код для многих функций C, или при недостаточном знакомстве с правилами использования стеков в Perl и прочими нюансами. XS позволяет составить краткое описание того, что должен делать связующий код, а компилятор XS *xsubpp* сделает все остальное.

Те, кому система XS кажется недостаточно удобной, могут использовать систему SWIG (Simplified Wrapper and Interface Generator) для автоматической генерации простых XSUB. Дополнительные сведения можно найти на *http://www.swig.org*.

Язык XS позволяет описать соответствие между функциями C и Perl. Он позволяет также создать функцию Perl, являющуюся оболочкой вокруг чистого кода C, который программист пишет самостоятельно. Если XS используется только для установления соответствия между C и Perl, объявление XSUB почти идентично объявлению функции C. В этом случае с помощью средства под названием *h2xs* (поставляемого с Perl) можно целиком транслировать файл заголовков C в соответствующий файл XS, осуществляющий связывание с функциями и макросами C.

Средство *xsubpp* создает конструкции, необходимые XSUB для обработки значений Perl, и связующий код, необходимый Perl для вызова XSUB.

В начале файла XS помещается любой код C, который требуется включить, часто представляющий собой всего лишь набор директив `#include`. После ключевого слова `MODULE` оставшаяся часть файла должна быть написана на «языке» XS и содержать директивы XS и определения XSUB. Вскоре мы рассмотрим пример файла XS целиком, но до того приведем простое определение XSUB, которое позволяет программе Perl обращаться к библиотечной функции C с именем *sin*(3). XSUB определяет тип возвращаемого значения (число с плавающей десятичной точкой, представляемое с двойной точностью), имя

функции и список аргументов (состоящий из одного аргумента, окрещенного x), а также типа аргумента (тоже число с двойной точностью):

```
double
sin(x)
    double x
```

Более сложные XSUB часто содержат другие фрагменты кода XS. Каждый раздел XSUB начинается с ключевого слова, за которым следует двоеточие, например, `INIT:` или `CLEANUP:`. Однако в первых двух строках XSUB всегда содержатся одни и те же данные: описание типа возвращаемого значения, имя функции и ее параметры. То, что непосредственно следует за ними, считается разделом `INPUT:`, если только явно не указано другое ключевое слово. Все ключевые слова описаны на странице руководства *perlxs*, которую следует прочесть, чтобы узнать обо всех возможностях применения XSUB.

Если система не позволяет динамически загружать библиотеки совместного доступа, то использование XSUB все-таки возможно, но необходимо статически компоновать XSUB с Perl, создавая новые исполняемые модули Perl. Механизм компиляции XSUB проверяет систему и, если это возможно, создает динамическую библиотеку, а в противном случае – статическую. Можно также выбрать вариант создания нового статически скомпонованного исполняемого модуля, в который включена эта статическая библиотека. (Это можно отложить на более позднее время, чтобы включить все созданные новые расширения в один исполняемый модуль.)

Если операционная система позволяет связывать библиотеки динамически, но вы хотите сделать статически скомпонованный исполняемый модуль, в последующих примерах можно запускать `make perl` вместо `make`. Для тестирования созданной дополнительной библиотеки в этом случае нужно запускать `make test_static` вместо `make test`.

Программе *xsubpp* необходимо также знать, как преобразовывать типы данных Perl в типы данных C. Как правило, она сама может это правильно определить, но для типов данных, определенных пользователем, может потребоваться помочь ей выйти из затруднения, указав преобразование в файле *typemap*. Преобразования, осуществляемые по умолчанию, заданы в *ПУТЬ-К-PERLLIB/ExtUtils/typemap*.

Файл typemap разбит на три секции. В первой секции, имеющей метку `TYPEMAP`, компилятору сообщается, какие фрагменты кода в последующих двух секциях следует использовать для установления соответствия между типами C и величинами Perl. Во второй секции, `INPUT`, содержится код C, определяющий преобразование значений Perl в типы C. В третьей секции, `OUTPUT`, содержится код C, определяющий способ трансляции типов C в значения Perl.

Создание расширений

Правильно созданное расширение состоит из нескольких файлов: в одном содержится код XS, другие вспомогательные файлы позволяют Perl разо-

браться, что нужно делать с этим кодом XS. Все эти файлы можно создать вручную, но проще воспользоваться средством *h2xs*, создающим скелет расширения, который можно потом дополнить всем необходимым:

```
h2xs -A -n Mytest
```

Эта команда создает каталог с именем *Mytest*, возможно, в качестве подкаталога *ext/*, если таковой существует в текущем каталоге. В каталоге *Mytest* будет создано шесть файлов: *MANIFEST*, *Makefile.PL*, *Mytest.pm*, *Mytest.xs*, *test.pl* и *Changes*. Первые четыре из них мы опишем ниже.

MANIFEST

Файл *MANIFEST* содержит имена всех файлов, только что созданных в каталоге *Mytest*. Если вы хотите добавить к своему расширению другие файлы и намерены сделать его достоянием всего света, включите их имена в этот файл. Некоторые системы проверяют его, чтобы гарантировать комплектность дистрибутива.

Makefile.PL

Это программа Perl, генерирующая *Makefile* (передаваемый затем *make* или ее эквиваленту). *Makefile.PL* описан далее в разделе «Создание модулей CPAN» главы 22 «CPAN».

Mytest.pm

Посредством этого модуля пользователи будут загружать ваше расширение. Необходимо заполнить пустые места в скелете модуля, созданном утилитой *h2xs*:

```
package Mytest;

use strict;
use warnings;

require Exporter;
require DynaLoader;

our @ISA = qw(Exporter DynaLoader);
# Items to export into callers namespace by default. Note: do not export
# names by default without a very good reason. Use EXPORT_OK instead.
# Do not simply export all your public functions/methods/constants.
# [Перевод:
# Элементы, по умолчанию экспортируемые в пространство имен
# вызывающей программы. Замечание: без очень веских причин не
# экспортируйте имена по умолчанию, вместо этого используйте
# EXPORT_OK. Не следует просто экспортировать все свои
# открытые функции/методы/константы.]
our @EXPORT = qw(

);
our $VERSION = '0.01';

bootstrap Mytest $VERSION;

# Preloaded methods go here.
```

```
# [Перевод:
# Здесь располагаются предварительно загружаемые (preloaded) методы.]

# Autoload methods go after __END__, and are processed by the autosplit program.
# [Перевод:
# Методы автоматической загрузки следуют после __END__ и
# обрабатываются программой autosplit.]

1;
__END__

# Below is the stub of documentation for your module. You better edit it!
# [Перевод:
# Далее находится заглушка для документации по вашему модулю.
# Отредактируйте ее!]
```

Для большинства модулей расширений требуются расширения `Exporter` и `DynaLoader`. После задания `@ISA` (для наследования) и `@EXPORT` (чтобы сделать функции доступными в пакете, использующем модуль) код инициализации сообщает Perl о необходимости произвести начальную загрузку кода XS. В результате Perl динамически компонует библиотеку совместного доступа с процессом *perl* на этапе исполнения.

Mytest.xs

Файл *Mytest.xs* содержит XSUB, в которых Perl сообщается о том, как передавать данные откомпилированным программам C. Изначально *Mytest.xs* выглядит примерно так:

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

MODULE = Mytest                PACKAGE = Mytest
```

Отредактируем файл XS, добавив в его конец следующие строки:

```
void
hello()
    CODE:
        printf("Hello, world!\n");
```

В результате запуска `perl Makefile.PL` будет создан *Makefile*, необходимый для *make*:

```
% perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for Mytest
```

После запуска *make* будет выведено примерно следующее (некоторые длинные строки для ясности укорочены, а некоторые посторонние строки удалены):

```
% make
umask 0 && cp Mytest.pm ./blib/Mytest.pm
```

```
perl xsubpp -typemap typemap Mytest.xs >Mytest.tc && mv Mytest.tc Mytest.c
cc -c Mytest.c
Running Mkbootstrap for Mytest ()
chmod 644 Mytest.bs
LD_RUN_PATH="" ld -o ./blib/PA-RISC1.1/auto/Mytest/Mytest.sl -b Mytest.o
chmod 755 ./blib/PA-RISC1.1/auto/Mytest/Mytest.sl
cp Mytest.bs ./blib/PA-RISC1.1/auto/Mytest/Mytest.bs
chmod 644 ./blib/PA-RISC1.1/auto/Mytest/Mytest.bs
Manifying ./blib/man3/Mytest.3
```

Мы считаем, что компиляцию и сборку программ Perl осуществляет посредством программы с именем *make*. Вместо использования в этих примерах *make* можно указать ту программу *make*, которая указана в конфигурации Perl. Узнать, что это за программа, можно с помощью команды:

```
% perl -V:make
```

При запуске *make* в текущем рабочем каталоге создается каталог *blib* (от «build library»). В этот каталог будет помещена создаваемая библиотека совместного доступа. При наличии уверенности, что все сделано правильно, можно установить ее из этого каталога туда, где она должна в итоге располагаться. До того момента необходимо явно добавить каталог *blib* в массив Perl `@INC` с помощью модуля `ExtUtils::testlib`. Теперь, если создать файл *hello* со следующим содержанием:

```
use ExtUtils::testlib;   # добавляет каталоги blib/* в @INC
use Mytest;
Mytest::hello();
```

то можно «прорваться» из Perl в C:

```
% perl hello
Hello, world!
```

После того как расширение будет полностью закончено и пройдет все проверки, можно будет установить его с помощью `make install`.

Нам потребуется доступ к библиотеке Perl с разрешением на запись. (При отсутствии такого разрешения можно задать другой каталог, как показано в разделе «Установка модулей CPAN» в главе 22.)

Ввод и вывод в XSUB

Продолжая предыдущий пример, добавим вторую подпрограмму XSUB, которая принимает на входе одно число и возвращает 1, если число четное, или 0, если нечетное:

```
int
is_even(x)
        int x
    CODE:
```

```
        RETVAL = (x % 2 == 0);
    OUTPUT:
        RETVAL
```

Список выходных параметров находится в самом конце функции, сразу после директивы `OUTPUT:`. Использование `RETVAL` указывает Perl, что вы хотите послать это значение в качестве значения, возвращаемого XSUB. Для того чтобы функция модифицировала свой входной параметр, следовало указать `x` вместо `RETVAL`.

Мы можем перекомпилировать нашу новую библиотеку совместного доступа, пройдя те же шаги, что и раньше, создавая *Makefile* из файла *Makefile.PL* и запуская *make*.

Чтобы убедиться, что наше расширение работает, мы сначала создадим набор тестов в *test.pl*. Этот файл создается *h2xs* в подражание тестовому сценарию, который имеется в самом Perl. Внутри этого сценария можно запускать тесты для проверки того, что расширение работает правильно, и выводящие `ok`, когда это так, и `not ok`, когда это не так. Измените команду print в блоке `BEGIN` файла *test.pl* на `print "1..4\n";` и добавьте в конец файла следующий код:

```
print Mytest::is_even(0) == 1 ? "ok 2" : "not ok 2", "\n";
print Mytest::is_even(1) == 0 ? "ok 3" : "not ok 3", "\n";
print Mytest::is_even(2) == 1 ? "ok 4" : "not ok 4", "\n";
```

Тестовый сценарий будет выполнен при вводе `make test`.

Использование функций во внешней библиотеке C

В двух предыдущих примерах мы не полагались на какой-либо код C вне файла XS. Сейчас мы воспользуемся некоторыми функциями из математической библиотеки C:

```
void
round(arg)
        double  arg
    CODE:
        if (arg > 0.0) {
                arg = floor(arg + 0.5);
        } else if (arg < 0.0) {
                arg = ceil(arg - 0.5);
        } else {
                arg = 0.0;
        }
    OUTPUT:
        arg
```

Обратите внимание, что функция `round`, которую мы определили выше, не возвращает значения, а вместо этого изменяет значение переданного ей аргумента.

Функции *floor*(3) и *ceil*(3) входят в состав математической библиотеки C. Компилируя программу на C и собираясь скомпоновать ее с математической библиотекой, следует добавить в командную строку `-lm`; то же самое мы делаем в строке `LIBS` в файле *Makefile.PL*:

```
'LIBS'  => ['-lm'],   # Компоновать с математической библиотекой
```

Сгенерируйте *Makefile* и запустите *make*. Измените блок `BEGIN`, чтобы выполнить девять тестов, и добавьте в *test.pl* следующее:

```
$i = -1.5; Mytest::round($i); print $i == -2.0 ? "ok 5" : "not ok 5", "\n";
$i = -1.1; Mytest::round($i); print $i == -1.0 ? "ok 6" : "not ok 6", "\n";
$i = 0.0; Mytest::round($i); print $i == 0.0 ? "ok 7" : "not ok 7", "\n";
$i = 0.5; Mytest::round($i); print $i == 1.0 ? "ok 8" : "not ok 8", "\n";
$i = 1.2; Mytest::round($i); print $i == 1.0 ? "ok 9" : "not ok 9", "\n";
```

Теперь при запуске `make test` должно быть выведено, что все девять тестов прошли нормально.

В поставляемой с Perl документацией *perlxstut* есть еще несколько примеров расширений Perl, в том числе пример использования *h2xs* для автоматического создания целой библиотеки C, доступной для Perl.

Вызов Perl из C

Можно получить доступ к интерпретатору Perl из C путем *встраивания* Perl в программу C. Поскольку Perl сам является программой C, встраивание заключается в том, чтобы взять важные фрагменты кода Perl и интегрировать их со своей программой.

Заметим, что во встраивании нет необходимости, если единственной нашей целью является использование самостоятельной программы Perl и нет возражений против применения для этого отдельного процесса. Обмен данными между программой C и любой внешней программой Perl можно организовать посредством такой функции, как *popen*(3) из C, – точно так же, как можно использовать в Perl `open(PIPE, "| program")` или модули `IPC::Open2` и `IPC::Open3` для обмена данными между программой Perl и любой другой программой. Но для того чтобы исключить накладные расходы, связанные с запуском отдельного процесса, можно встроить интерпретатор в C-программу.

При разработке приложений, выполняемых длительное время (скажем, для встраивания в веб-сервер), хорошим решением является поддержка одного постоянного интерпретатора вместо многократного создания и уничтожения интерпретаторов. Главным основанием для этого служат соображения скорости, поскольку Perl в этом случае будет загружаться в память только один раз. В случае применения постоянного интерпретатора Perl модуль Apache `mod_perl` позволяет избежать новой загрузки Perl в память всякий раз, когда кто-то обращается к веб-странице Apache. На странице руководства *perlembed* приводится пример постоянного интерпретатора, а также того, как программа на Perl может управлять одновременно несколькими интерпретаторами (еще одно преимущество для веб-серверов).

Компиляция встроенных программ

При встраивании Perl в C программа на C обычно размещает в памяти, «запускает» и освобождает объект `PerlInterpreter`, представляющий собой конструкцию `struct` в C, определенную в библиотеке *libperl*, созданной в процессе конфигурирования Perl для данной системы. Каталог, в котором находится библиотека *libperl* (а также *EXTERN.h* и *perl.h*, которые вам тоже понадобятся), в разных системах может быть различным. Определить название этого каталога можно так:

```
% perl -MConfig -e "print $Config{archlib}"
```

Необходимо компилировать свою программу точно так же, как компилировался исполняемый модуль *perl*. Во-первых, нужно узнать, какой компилятор C использовался при создании Perl на вашей машине. Например:

```
% perl -MConfig -e "print $Config{cc}"
```

Можно определить, что еще нужно поместить в командную строку, с помощью стандартного модуля `ExtUtils::Embed`. Если программа C называется *interp.c*, а компилятором C был *cc*, ее можно скомпилировать для встраивания с помощью такой команды:

```
% cc -o interp interp.c `perl -MExtUtils::Embed -e ccopts -e ldopts`
```

Добавление интерпретатора Perl в программу на C

Оказывается, что *perl* (C-программа) дает хороший пример встраивания Perl (языка), поэтому простую иллюстрацию встраивания можно найти в файле *miniperlmain.c*, входящем в исходный код Perl. Вот непереносимый вариант *miniperlmain.c*, в котором содержится сущность встраивания:

```
#include <EXTERN.h)               /* from the Perl distribution     */
#include <perl.h>                 /* from the Perl distribution     */

static PerlInterpreter *my_perl;  /***    The Perl interpreter    ***/

int main(int argc, char **argv, char **env)
{
    my_perl = perl_alloc();
    perl_construct(my_perl);
    perl_parse(my_perl, NULL, argc, argv, (char **)NULL);
    perl_run(my_perl);
    perl_destruct(my_perl);
    perl_free(my_perl);
}
```

При компиляции этого файла с помощью приведенной выше команды мы получим возможность использовать *interp* как обычный интерпретатор Perl:

```
% interp -e "printf('%x', 3735928559)"
deadbeef
```

Можно также выполнять команды Perl, находящиеся в файле, задавая имя этого файла в argv[1] перед вызовом perl_run.

Вызов подпрограммы Perl из C

Если программа Perl содержит подпрограмму, которую необходимо вызвать из программы на C, то можно создать интерпретатор Perl, а затем использовать одну из функций, имена которых начинаются с call_ и документация по которым имеется на странице руководства *perlcall*, для вызова подпрограммы. Допустим, что у нас есть такая программа Perl с именем *showtime.pl*:

```
print "Это не будет выведено.";

sub showtime {
    print time;
}
```

В этом примере мы воспользуемся функцией call_argv для вызова подпрограммы showtime из программы на C с именем *showtime.c*:

```
#include <EXTERN.h>
#include <perl.h>

static PerlInterpreter *my_perl;

int main(int argc, char **argv, char **env)
{
    char *args[] = { NULL };
    my_perl = perl_alloc();
    perl_construct(my_perl);
    perl_parse(my_perl, NULL, argc, argv, NULL);

    /*** пропуск perl_run() ***/

    call_argv("showtime", G_DISCARD | G_NOARGS, args);

    perl_destruct(my_perl);
    perl_free(my_perl);
}
```

Здесь мы предполагаем, что showtime является подпрограммой Perl, которая не принимает аргументы (что указано с помощью G_NOARGS) и для которой можно игнорировать возвращаемое значение (что указано с помощью G_DISCARD). Об этих и других флагах сказано в *perlcall*. Мы компилируем и запускаем *showtime* следующим образом:

```
% cc -o showtime showtime.c `perl -MExtUtils::Embed -e ccopts -e ldopts`
% showtime showtime.pl
963852741
```

В данном конкретном случае мы не запускаем perl_run, хотя вообще это считается правильным, с тем чтобы в нужное время выполнились методы DESTROY и блоки END.

Для передачи аргументов подпрограмме Perl можно добавить строки к завершающемуся NULL списку args, передаваемому call_argv. Чтобы использовать другие типы данных или рассматривать возвращаемые значения, потребуется поработать со стеком Perl. Мы коснемся этого чуть позже, а детали и подробности можно найти на странице руководства *perlcall*, поставляемой с Perl.

Выполнение команды Perl из C

Perl предоставляет две функции для выполнения фрагментов кода Perl, eval_sv и eval_ pv, которые описаны на странице руководства *perlapi*. Можно спорить, что это единственные программы, которые когда-либо вам понадобятся для выполнения кода Perl из своих программ на C. Выполняемый код может иметь произвольную длину, содержать несколько команд и использовать use, require или do для подключения других файлов Perl.

Функция eval_ pv позволяет выполнять отдельные строки Perl и извлекать переменные для перевода их в типы C. В следующей программе, *string.c*, выполняются три строки Perl; из первой извлекается int, из второй – float и из третьей – char*:

```
#include <EXTERN.h>
#include <perl.h>

static PerlInterpreter *my_perl;

main (int argc, char **argv, char **env)
{
   STRLEN n_a;
   char *embedding[] = { "", "-e", "0" };
   my_perl = perl_alloc();
   perl_construct( my_perl );

   perl_parse(my_perl, NULL, 3, embedding, NULL);
   perl_run(my_perl);

   /** Обработать $a как целое число **/
   eval_pv("$a = 3; $a **= 2", TRUE);
   printf("a = %d\n", SvIV(get_sv("a", FALSE)));

   /** Обработать $a как действительное число **/
   eval_pv("$a = 3.14; $a **= 2", TRUE);
   printf("a = %f\n", SvNV(get_sv("a", FALSE)));

   /** Обработать $a как строку **/
   eval_pv("$a = 'relreP kcaH rehtonA tsuJ'; $a = reverse($a);", TRUE);
   printf("a = %s\n", SvPV(get_sv("a", FALSE), n_a));

   perl_destruct(my_perl);
   perl_free(my_perl);
}
```

Все функции, в имена которых входит Sv, преобразуют скаляры Perl в типы C. Они описаны в страницах руководства *perlguts* и *perlapi*. Если скомпилировать и выполнить эту программу, то можно увидеть результаты применения SvIV для создания int, SvNV для создания float и SvPV для создания строки C:

```
a = 9
a = 9.859600
a = Just Another Hack Perler
```

В предыдущем примере мы создали глобальную переменную для временного хранения полученного значения вычисленного выражения. Можно также (и, как правило, это лучше) использовать значение, возвращаемое eval_pv, а не отбрасывать его:

```
SV *val = eval_pv("reverse 'relreP kcaH rehtonA tsuJ'", TRUE);
printf("%s\n", SvPV(val,n_a));
```

Страница руководства *perlembed*, поставляемая с Perl, содержит демонстрацию применения eval_sv, в которой показано, как использовать возможности регулярных выражений Perl из программы на C.

Игры со стеком Perl из C

Пытаясь объяснить, что такое стеки, авторы большинства учебников по информатике[1] рассказывают историю о стоящих на пружинах тарелках в кафетерии: последнее, что мы протолкнули на стек, будет первым, что мы из него вытолкнем. Для наших целей это годится: программа C будет проталкивать какие-то аргументы в «стек Perl», зажмуривать глаза, пока совершаются какие-то чудеса, а затем выталкивать результаты – значение, возвращаемое подпрограммой Perl, – из стека.

Мы приведем здесь пример без особого его разъяснения. Чтобы действительно понять, что происходит, вам потребуется знать, как осуществлять преобразование между типами C и типами Perl с помощью newSViv и sv_setnv, и newAV, и всех подобных им, описанных на страницах руководства *perlguts* и *perlapi*. Затем надо будет прочитать *perlcall*, чтобы узнать, как работать со стеком Perl.

Поскольку в C нет встроенной функции для возведения в степень целых чисел, давайте сделаем оператор Perl ** доступным программе на C. (Это не так полезно, как кажется, поскольку в Perl оператор ** реализован через функцию C *pow*(3).) Сначала мы создадим функцию экспоненты в библиотечном файле *power.p l*:

```
sub expo {
    my ($a, $b) = @_;
    return $a ** $b;
}
```

Теперь создадим программу C, *power.c*, с функцией PerlPower, которая проталкивает два аргумента в стек, вызывает функцию expo и выталкивает обратно возвращаемое значение:

```
#include <EXTERN.h>
#include <perl.h>
```

[1] И этой случайно попавшейся книги по Perl.

```
static PerlInterpreter *my_perl;

/* "Настоящие программисты могут использовать ассемблерный код в любом языке." */

static void
PerlPower(int a, int b)
{
  dSP;                            /* инициализировать указатель стека          */
  ENTER;                          /* все создаваемое далее                     */
  SAVETMPS;                       /*    является временной переменной          */
  SAVETMPS;                       .* запомнить указатель стека                 *.
  XPUSHs(sv_2mortal(newSViv(a))); /* вытолкнуть на стек основание              */
  XPUSHs(sv_2mortal(newSViv(b))); /* вытолкнуть на стек показатель             */
  PUTBACK;                        /* сделать локальный указатель стека         */
                                  /*    глобальным                             */
  call_pv("expo", G_SCALAR);      /* вызвать функцию                           */
  SPAGAIN;                        /* обновить указатель стека                  */
                                  /* вытолкнуть возвращаемое значение со стека */
  printf ("%d to the %dth power is %d.\n", a, b, POPi);
  PUTBACK;
  FREETMPS;                       /* освободить возвращаемое значение          */
  LEAVE;                          /* ...и "смертные" аргументы XPUSHed         */
}
int main (int argc, char **argv, char **env)
{
  char *my_argv[] = { "", "power.pl" };

  my_perl = perl_alloc();
  perl_construct( my_perl );

  perl_parse(my_perl, NULL, 2, my_argv, (char **)NULL);
  perl_run(my_perl);

  PerlPower(3, 4);                     /*** Вычислить 3 ** 4 ***/

  perl_destruct(my_perl);
  perl_free(my_perl);
}
```

Скомпилировать *power.c* в *power* можно так:

```
% cc -o power power.c `perl -MExtUtils::Embed -e ccopts -e ldopts`
% power
3 to the 4th power is 81.
```

Теперь ваша программа *power* тоже может прохлаждаться и воображать.

Мораль «сей басни»

Иногда можно написать более быстрый код на C, но всегда можно быстрее написать код на Perl. Поскольку один язык можно использовать из другого, сочетайте силу каждого так, как считаете подходящим. (И передайте драконам, что мы сказали: «Привет!»)

IV

Perl как культура

22

CPAN

CPAN (Comprehensive Perl Archive Network – архив Perl) является центральным хранилищем всего, что касается Perl. В нем содержится коллективная мудрость всего сообщества Perl: сотни модулей и сценариев Perl, документация, которой хватило бы на несколько книг, и полный дистрибутив Perl. Если что-то написано на Perl, приносит пользу и бесплатно, то, вероятно, оно имеется в CPAN. Зеркальные серверы CPAN разбросаны по всему свету, и ближайший из них вы найдете через редиректор CPAN на *http://www.perl.com/CPAN*. Редиректор запомнит, какое зеркало вы выбрали, и при следующем посещении *http://www.perl.com/CPAN/* (обратите внимание на косую черту в конце) вы будете автоматически переадресованы на это зеркало. Вы можете также начать с *www.cpan.org*. Интерфейс другой, но содержание то же самое.

Попав в главный каталог CPAN, вы увидите некоторые подкаталоги:

authors

Этот каталог содержит многочисленные подкаталоги, по одному для каждого, кто предоставил программное обеспечение. Например, захотев найти замечательный модуль CGI,[1] который разработал Линкольн Штейн (Lincoln Stein), и зная, что этот модуль действительно написал он, можно посмотреть каталог *authors/Lincoln_Stein*. Не зная, кем написан модуль, можно было бы поискать в каталоге *modules*, описываемом ниже.

---

[1] Сейчас он входит в стандартный дистрибутив Perl.

doc

В этом каталоге содержатся все виды документации по Perl, в том числе официальные страницы руководства по Perl в нескольких разных видах и форматах, например, в текстовом, HTML, PostScript и родном для Perl формате pod, документированном в главе 26 «Документация в формате pod».

modules

Этот каталог содержит модули, написанные на Perl или комбинации Perl и C. См. далее описание каталога *modules*.

ports

Этот каталог содержит исходный код и в некоторых случаях исполняемые образы для Perl на платформах, отличных от UNIX, которые не поддерживаются непосредственно стандартным дистрибутивом или для которых известную трудность составляет получение компилятора. Эти портированные приложения являются собственными проектами авторов и не всегда работают точно так, как описано в этой книге. В настоящее время есть не много систем, для которых требуются специальные действия при переносе. Указатель, имеющийся в этом каталоге, в любом случае интересно посмотреть, поскольку в нем содержатся сведения о том, с какого времени каждый поставщик операционной системы начал предоставлять Perl.

scripts

Этот каталог содержит небольшую коллекцию разнообразных программ на Perl, собранных со всего мира. Они полезны как отдельные программы и, возможно, в качестве примеров (хотя этот код не проходил через контроль качества). В настоящее время там находится не много программ, но мы надеемся, что этот раздел со временем обогатится. Проект инструментальных средств Perl Power Tools (PPT) тоже находится здесь. Целью PPT является воссоздание всех стандартных утилит Unix в Perl. Это уже сделано в отношении большинства стандартных утилит и части нестандартных.

src

В этом каталоге вы найдете исходный код для стандартного дистрибутива Perl. Фактически для двух стандартных дистрибутивов Perl. Один из них помечен как *stable* (*стабильный*), а другой как *devel* (*разрабатываемый*). (Детали разъясняются на странице содержания этого каталога.) На самом деле это только ссылки на соответствующие версии. В период написания этой книги *stable.tar.gz* была символической ссылкой на *perl-5.6.0.tar.gz*,[1] но когда эти строки увидит читатель, она, скорее всего, будет

[1] Схема нумерации такова, что если второе число в номере версии четное, то это сопровождаемая версия, если нечетное, то это версия разработчиков. Расширение *.tar.gz*, иногда записываемое как *.tgz*, указывает на стандартный для Unix-систем формат архива *tar*, сжатого GNU-zip, часто называемого «tarball».

указывать на более высокий номер версии. Этот гигантский файл содержит полный исходный код и документацию для Perl. Конфигурирование и установка должны быть относительно просты для большинства платформ. Если нет, см. *ports* выше.

Каталог modules архива CPAN

Хотя CPAN содержит полный исходный код Perl плюс несколько исполняемых дистрибутивов для систем, лишенных компиляторов C, а также ряд программ, более всего CPAN известен своей коллекцией модулей.

Говоря «модули», мы подразумеваем три вещи: модули, написанные на 100-процентном чистом Perl (описываемые в главе 11 «Модули» и главе 12 «Объекты»), расширения (модули, зависящие от кода C, описанные в главе 21 «Внутри и снаружи») и прагмы (модули, содержащие специальные инструкции для компилятора Perl, описываемые в главе 31 «Модули прагм»). В CPAN есть также *связки* (*bundles*) модулей. Связки представляют собой совокупности модулей, определенным образом взаимодействующих между собой и обычно являющиеся результатом желания разработчика предоставить решение «под ключ» для группы задач. Если один модуль зависит от другого (и, возможно, от конкретной его версии), то разработчики часто создают из модулей связку. См. например `Bundle-XML`.

Один из способов просмотреть модули CPAN – посетить сайт *http://search.cpan.org*, предоставляющий интерфейс к поисковой машине CPAN. Другой способ состоит в том, чтобы зайти в каталог *modules* на вашем местном (зональном) зеркале CPAN, где вы увидите три подкаталога: *by-authors*, *by-category* и *by-module*. Каталог *by-module* может оказаться самым полезным, если ваш броузер поддерживает поиск, хотя (к сожалению) некоторые модули есть только в каталогах своих авторов. Для поиска по категориям предоставляется следующий выбор:

Perl core modules (базовые модули Perl), language extensions (расширения языка), documentation tools (средства документации)

Сюда входят прагмы и другие стандартные модули, модули, позволяющие писать на Perl иначе, обеспечивая дополнительные возможности, модули, относящиеся к компилятору Perl, фильтры исходного кода и модули, относящиеся к формату pod документации Perl. В эту категорию входят также модули для генерации байт-кода Java.

Development support (поддержка разработки)

В эту категорию входят модули для создания модулей и изучения того, как Perl выполняет программы.

Operating system interfaces and hardware drivers (интерфейсы операционных систем и драйверы аппаратного обеспечения)

Здесь вы найдете модули для взаимодействия с такими «странными» объектами, как операционные системы, PalmPilot и последовательные порты.

Networking (сетевое взаимодействие), device control (управление устройствами), interprocess communication (межпроцессное взаимодействие)

Сюда входят модули, реализующие сетевые протоколы, манипулирующие данными в сетях, управляющие модемами и бытовой техникой.

Data types and data type utilities (типы данных и утилиты типов данных)

В этой категории имеются модули для математических вычислений, статистики, алгоритмов, структур данных (и их постоянного хранения), даты и времени, объектно-ориентированного программирования, PDL (Perl Data Language, для серьезных вычислений) и POE (Perl Object Environment, объектно-ориентированный, управляемый событиями планировщик).

Database interfaces (интерфейсы баз данных)

Здесь вы найдете модули, позволяющие работать из Perl с несколькими десятками СУБД, по большей части с помощью системы Perl DBI. В это число входят модули DBD для конретных баз данных.

User interfaces, character and graphical (пользовательские интерфейсы символьные и графические)

Сюда входят модули для работы с терминалами пользователя (редактирование командной строки и символьная графика в стиле *curses*(3)), а также Perl/Tk и связи с Gtk, Gnome, Sx и Qt для создания собственных GUI на Perl.

Interfaces to or emulations of other programming languages (интерфейсы к другим языкам программирования или эмуляция их)

В эту категорию попадают модули, позволяющие применять в Perl другие языки программирования или сделать вид, что Perl является чем-то таким, чем он на самом деле не является. Использование Perl из C или C из Perl рассмотрено в главе 21 «Внутри и снаружи».

Filenames, filesystems, and file locking (имена файлов, файловые системы и блокировка файлов)

Сюда входят модули для контролирования, создания, блокировки и прочих манипуляций с файлами и каталогами.

String processing, language text processing, parsing, and searching (обработка строк, обработка исходного текста, синтаксический анализ и поиск)

В эту категорию попадают модули для обработки текста: разбиения на слоги, переноса, синтаксического анализа, выделения основ и поиска. Сюда входят модули для работы с PostScript, шрифтами, XML и RTF.

Option, argument, parameter, and configuration file processing (обработка опций, аргументов, параметров и файлов конфигурации)

Эта категория содержит модули для обработки параметров командной строки (например, -x в *myperlprog -x*) и управления файлами конфигурации (например, dot-файлами).

Internationalization and locale (многоязычность и настройки локализации)

Сюда входят модули для настройки вашей программы Perl на конкретные страну и язык.

Authentication, security, and encryption (аутентификация, безопасность и шифрование)

Здесь вы найдете модули для работы с паролями пользователей, вычисления контрольных сумм сообщений, шифрования данных и аутентификации пользователей.

World Wide Web, HTML, HTTP, CGI, MIME

В этой категории имеются модули, позволяющие создавать веб-страницы с использованием CGI, веб-роботов и системы управления содержанием для Сети. Другие модули позволят вам управлять файлами cookie, производить разбор сообщений HTML и MIME и работать с веб-кэшами. Здесь есть также специальный раздел модулей Perl, встраиваемых в веб-сервер Apache.

Server and daemon utilities (утилиты серверов и демонов)

Сюда входят модули для создания сетевых серверов и серверов событий.

Archiving, compression, and conversion (архивация, сжатие и преобразование)

Здесь вы сможете найти модули для zip- и tar-архивации файлов, а также преобразования между различными форматами файлов (даже в формат файлов Apple II).

Images, pixmap, and bitmap manipulation, drawing, and graphing (работа с изображениями, пикселами и растрами, черчение и создание диаграмм)

В этой категории содержатся модули для создания диаграмм, GIF-файлов, VRML и работы с Gimp.

Mail and Usenet news (электронная почта и телеконференции)

В этой категории вы найдете модули для отправки, получения и фильтрации почты сообщений телеконференций.

Control flow utilities (утилиты управления последовательностью выполняемых команд)

Модули для выполнения кода Perl в определенные моменты времени.

Filehandle, directory handle, and input/output stream utilities (утилиты для указателей файлов, каталогов и потоков ввода-вывода)

Здесь находятся модули для ввода и вывода из файлов, в том числе журнальных (log-файлов). Сюда входят все модули `IO::` и модуль `Expect` для автоматизации общения с сетевыми сервисами или другими интерактивными программами.

Microsoft Windows modules (модули для Microsoft Windows)

Сюда входят модули для работы с реестром Windows, ASP, ODBC, OLE и другими специальными технологиями для Windows.

Miscellaneous modules (разные модули)

Модули для астрономии, биологии, химии, проверки достоверности (и недостоверности) кредитных карт, погашения ссуд, аудио, видео, MIDI, погоды и игр.

Использование модулей CPAN

Большинство модулей в CPAN находятся в сжатом архиве tar. То есть они имеют расширение *.tar.gz* и распаковываются в каталог, содержащий код модуля и вспомогательные файлы, в число которых обычно входят *README* и *Makefile.PL*.

Чтобы сделать модули CPAN доступными вашим программам, нужно пройти четыре шага: декомпрессию, разархивирование, компиляцию и установку. Действия на каждом из этих этапов зависят от операционной системы и устанавливаемого модуля, поэтому мы не можем дать абсолютного рецепта, пригодного на все случаи жизни. При возникновении сомнений читайте файлы *README* и *INSTALL*, которые, как следует надеяться, распространяются с модулем. Прочтите также страницу руководства *perlmodinstall*.

Однако возможно, что вам не придется думать о процедуре установки, если вы воспользуетесь модулем `CPAN` (поставляемым в связке с дистрибутивом Perl) или PPM (Perl Package Manager – администратором пакетов Perl, поставляемым в связке с дистрибутивом Perl ActiveState). Чтобы иметь возможность работать с модулем `CPAN` (не путайте с собственно CPAN), введите в командной строке:

```
% perl -MCPAN -e "shell"
```

и начните процесс конфигурирования. Ответив на ряд вопросов о том, как вы хотите извлекать файлы, можете установить модуль, введя в оболочке модуля `CPAN`:

```
install Some::Module
```

или в обычной командной строке:

```
% perl -MCPAN -e "install 'Some::Module'"
```

Если вы лишены удобств, предоставляемых модулем `CPAN` или PPM, придется вручную пройти этапы, описанные в следующих разделах. Инструкции приводятся для Unix, Windows и Macintosh; для других операционных систем обратитесь к странице руководства *perlmodinstall*.

Декомпрессия и распаковка модулей CPAN

Большая часть тысяч утилит, имеющихся в CPAN, сжата, чтобы занимать меньше места. Получив архив с модулем, вы сначала должны превратить его в дерево каталогов на вашей машине путем декомпрессии («unzipping»)

и распаковки. В Unix для этого применяются *gzip* и *tar*. (Во многих системах *tar* выполняет обе задачи.) В Windows *WinZip* одновременно декомпрессирует и распаковывает архивный файл. На Macintosh можно использовать *StuffIt* и *DropStuff*, *MacGzip* или *suntar*.

Компиляция модулей CPAN

Небольшая часть модулей CPAN поступает с кодом C, который нужно откомпилировать для вашей системы, что, естественно, вызывает неудобства на системах, в которых отсутствует компилятор C. Стандартной процедурой для компиляции модуля CPAN (с кодом C или без него) являются следующие три команды, выполняемые из командной строки. (Если у вас нет командной строки или эквивалента *make*, придется прибегнуть к более суровым мерам, которые зависят от системы. У пользователей Windows есть командная строка, но им может потребоваться вместо *make* утилита *nmake*.)

```
% perl Makefile.PL

% make

% make test
```

Команда *perl Makefile.PL* попытается создать *Makefile*, с помощью которого последующая команда *make* определяет, какие утилиты и в каком порядке должны быть скомпилированы. Завершающая команда, *make test*, запускает набор тестов, которые, как надо надеяться, автор включил в комплект.

Установка модулей CPAN в библиотеку Perl

Те, кто выполнил предшествующие шаги, должны были получить модуль – откомпилированный и протестированный, но еще не установленный в библиотеку Perl. При установке Perl в вашей системе был создан каталог *lib* для хранения модулей, прагм и связанных с ними файлов. Это и есть библиотека Perl, которая обычно находится в каталоге типа */usr/local/lib/perl5* на системах Unix и по умолчанию *C:\PERL\LIB* на системах Windows. Модули, установленные после сборки Perl, хранятся в подкаталоге *site_perl* библиотеки Perl. Имена всех своих библиотечных каталогов (и кучу прочей информации) вы можете увидеть, сказав:

```
% perl -V
```

Чтобы установить модуль, введите:

```
% make install
```

Обычно даже для установки модулей в специфические для вашей машины библиотечные каталоги Perl требуются права суперпользователя.

Немного потрудившись, вы сможете установить модуль в каталог, находящийся вне вашей библиотеки Perl (например, в свой исходный каталог).

Вместо обычного ввода *perl Makefile.PL* для создания *Makefile* можно использовать такое «заклинание»:

```
% perl Makefile.PL LIB=/my/dir/perllib  \
        INSTALLMAN1DIR=/my/dir/man/man1 \
        INSTALLMAN3DIR=/my/dir/man/man3 \
            INSTALLBIN=/my/dir/bin      \
         INSTALLSCRIPT=/my/dir/scripts
```

В результате модули будут установлены в каталог */my/dir/perllib*, а остальные файлы – туда, куда нужно. (Тот, кому кажется, что слишком много приходится вводить с клавиатуры, может написать маленькую программу на Perl, которая будет делать это вместо него. Perl удобно использовать для таких вещей.)

После этого можно заставить Perl осуществлять поиск модулей в специальном каталоге, добавив:

```
use lib "/my/dir/perllib";
```

прежде чем программа попытается загрузить модуль. Можно также установить этот каталог в переменной окружения `PERL5LIB` или использовать ключ Perl `-I`. Примеры выполнения этого см. в описании директивы `use lib` в главе 31 «Модули прагм».

Создание модулей CPAN

Если у вас есть модуль, который, как вам кажется, может быть полезен другим, подумайте о том, чтобы загрузить его в CPAN и усовершенствовать таким способом наш мир. Сервер, который обрабатывает представляемые новые модули, носит название PAUSE (Perl Authors Upload Server) и находится на *https://pause.kbx.de/pause/*. Прежде чем загружать свой модуль, следует получить на PAUSE учетную запись. Это приближает вас к получению статуса «зарегистрированного разработчика Perl».

Если вы называете себя зарегистрированным разработчиком Perl, то должны знать, как документировать свои модули. В Perl принято встраивать документацию внутрь исходного кода. (Благодаря этому вы ее никогда не потеряете.) Формат этой встроенной документации носит название «pod» (от Plain Old Documentation – простая старая документация) и описан в главе 26 «Документация в формате POD».

Следует подумать о том, чтобы модуль поддерживал безопасные потоки. См. главу 17 «Потоки».

Надо также немного побеспокоиться о том, не делает ли ваш маленький умный модуль чего-то такого, что может нанести урон безопасности людей, его использующих, поскольку у других людей могут быть Достаточно Веские Основания, чтобы больше беспокоиться о безопасности, чем это делаете вы (пока). Читайте в главе 23 «Защита данных» обо всем, что касается того, как

избежать ответственности за развязывание Третьей мировой войны и других подобных неприятностей.

Модули, предназначенные для распространения через CPAN, должны содержать программу Perl с именем *Makefile.PL*, которая генерирует *Makefile* и файл *README* с кратким описанием того, что делает модуль и как его устанавливать. *Makefile* рассчитывает также, что ваш модуль содержит и комплект тестов. Все эти файлы можно создать одновременно с помощью утилиты *h2xs*:

```
h2xs -X -n Foo::Bar
```

(Замените `-X` на `-A`, если вы компилируете модуль, в который входит XS. XS описывается в главе 21 «Внутри и снаружи».) Программа *h2xs* создает один каталог со «скелетами» файлов, которые вы должны «облечь плотью». Все подготовив, можете загрузить заархивированный каталог на PAUSE.

Файл *Makefile.PL*, создаваемый *h2xs*, будет выглядеть примерно так:

```
use ExtUtils::MakeMaker;
# See lib/ExtUtils/MakeMaker.pm for details of how to influence
# the contents of the Makefile that is written.
WriteMakefile(
    NAME         => 'Mytest',
    VERSION_FROM => 'Mytest.pm', # finds $VERSION
    LIBS         => [''],   # e.g., '-lm'
    DEFINE       => '',     # e.g., '-DHAVE_SOMETHING'
    INC          => '',     # e.g., '-I/usr/include/other'
);
```

Сперва *Makefile.PL* загружает модуль `ExtUtils::MakeMaker`. Функция `WriteMakefile` модуля `MakeMaker` имеет огромное множество опций (где «огромное множество» равно примерно 88), которые помогают определить, что произойдет после того, как пользователь скачает модуль из CPAN и введет с клавиатуры *perl MakeFile.PL*, чтобы начать его компиляцию. Что во всем этом радует, так это то, что разработчику доступна масса сведений о конфигурации (через модуль `Config` или специальную переменную `$^O`), которые помогут ему решить, как управлять `MakeMaker`, – ведь пользователь, предположительно, запустит *perl*, который будет использоваться позже. С другой стороны, `MakeMaker` хорош тем, что почти для всего находит приличные значения по умолчанию, поэтому вам может оказаться достаточно скелета файла, написанного *h2xs*, возможно, с одним-двумя изменениями. Подробнее об этом читайте в обширной электронной документации по `ExtUtils::MakeMaker`.

Создавая модуль Perl для всеобщего употребления, следует допускать, что версия Perl пользователя может отличаться от вашей. Всегда помещайте описание на английском языке любых зависимых объектов (конкретных версий Perl, системных требований, других модулей) в файл *README*. Однако даже этого может оказаться недостаточно, потому что тот, кто использует `CPAN` для автоматической загрузки и установки вашего модуля, может не увидеть предупреждения. Поэтому необходимо проверить эти зависимос-

ти в *Makefile.PL*. Вот как можно проверить, что тот, кто загрузил ваш модуль, использует Perl 5.6 или выше:

```
eval { require 5.6.0 }
    or die <<'EOD';
############
### Этот модуль требует подпрограмм, которые могут служить левым значением,что
### недоступно в версиях Perl до 5.6. Пожалуйста, сделайте обновление!
############
EOD
```

Внутреннее тестирование

Стандартные инструкции по установке модуля сообщают пользователю, что нужно запустить *make test* после компиляции модуля с помощью *make*. Поэтому включайте, пожалуйста, подходящий тестовый сценарий в каждый модуль, который вы помещаете в CPAN. Следует моделировать стиль `ok/not ok`, который Perl использует в собственном наборе тестов, чтобы пользователю было легко определить исход каждого пункта теста. На первых порах вам поможет файл *test.pl*, генерируемый *h2xs*. В главе 21 «Внутри и снаружи» приведены примеры тестов, которые можно добавить в конец *test.pl*.

Если вы наметили много позиций для тестирования, то попробуйте подражать набору тестов Perl путем создания подкаталога *t/* в каталоге модуля и добавления *t* к именам ваших тестовых сценариев. При запуске *make test* все тесты будут запущены автоматически.

Внешнее тестирование

Модули, помещаемые на CPAN, тестируются множеством добровольцев на различных платформах. Эти тестеры CPAN извещаются по почте о каждой новой загрузке и посылают в список ответы `PASS` (прошел), `FAIL` (не прошел), `NA` (не применим к данной платформе) или `UNKNOWN` (неизвестно), вместе с другими замечаниями. Почтовый список рассылки для тестеров CPAN *cpan-testers@perl.org*; результаты отправляются на *http://testers.cpan.org/*.

Конечно, это только предварительное тестирование. Настоящее тестирование начинается тогда, когда кто-то включит ваш маленький модуль в веб-сервер, который генерирует миллион страниц в день. Или использует ваш модуль при проектировании самолета, на котором в один прекрасный день вы полетите.

Так что вперед, плюньте на эти противные маленькие тесты. Посмотрим, что будет...

23

Защита данных

И в том случае, когда вы имеете дело с пользователем, сидящим за клавиатурой и вводящим команды, и в другом, когда кто-то передает вам информацию по сети, нужно осторожно обращаться с данными, поступающими в вашу программу, поскольку другой человек может, по злому умыслу или случайно, послать вам нечто такое, что причинит вред. Perl предоставляет специальный механизм проверки безопасности, который носит название *режима меченых данных* (*taint mode*), назначением которого является изоляция данных, поступивших в вашу программу извне, чтобы вы не смогли сделать с их помощью что-либо такое, чего вы делать не собирались. Например, обратившись по ошибке к файлу сомнительного происхождения, можно дописать запись к файлу паролей, думая при этом, что она дописывается к журнальному файлу. Механизм пометки освещен в разделе «Обработка ненадежных данных».

В многозадачной среде закулисные действия невидимых актеров могут повлиять на безопасность вашей собственной программы. Тот, кто полагает, что единолично владеет внешними объектами (в особенности, файлами), как если бы его процесс был единственным в системе, подвергает себя значительно более неочевидным опасностям, чем те, которые являются следствием непосредственной работы с данными или кодом сомнительного происхождения. Perl оказывает нам кое-какую помощь, выявляя ряд ситуаций, которые мы не можем контролировать, а для тех, которые мы можем контролировать, главное – понимать, какие подходы являются непроницаемыми для невидимого вмешательства. Эти вопросы обсуждаются в разделе «Обработка ошибок синхронизации».

Если данные, получаемые со стороны, оказываются фрагментом кода, который нужно выполнить, следует проявлять еще большую осторожность, чем с данными, полученными со стороны. Perl осуществляет проверку для перехвата скрытого кода, маскирующегося под данные, чтобы вы ненароком его не выполнили. Если вы все же хотите выполнить посторонний код, модуль `Safe` позволит поместить подозрительный код в карантин, где он не сможет причинить никакого вреда и, может быть, сделает что-нибудь полезное. Это темы раздела «Обработка ненадежного кода».

Обработка ненадежных данных

Perl облегчает безопасное программирование, даже когда вашу программу использует кто-то, кому можно доверять еще меньше, чем самой программе. Это значит, что некоторые программы должны давать ограниченные права своим пользователям и не давать других прав. В Unix в эту категорию попадают программы setuid и setgid, как и программы, работающие в режимах с разными привилегиями, в других операционных системах, поддерживающих такие понятия. Но и в тех системах, которые их не поддерживают, те же принципы применимы к сетевым серверам и любым программам, выполняемым на этих серверах (например, сценариям CGI, процессорам почтовых списков рассылки и демонам, перечисленным в */etc/inetd.conf*). Все такие программы требуют более тщательного исследования, чем обычные.

Даже программы, выполняемые из командной строки, иногда являются хорошими кандидатами на режим проверки меченых данных, особенно если предполагается, что их будет выполнять привилегированный пользователь. Программы, обрабатывающие не заслуживающие доверия данные, например, такие, которые генерируют статистические данные из файлов журналов или используют `LWP::*` и `Net::*` для получения удаленных данных, следует запускать с явным указанием проверки данных на чистоту данных; программы, не проявляющие осторожность, рискуют превратиться в «троянского коня». Поскольку, идя на риск, программы не получают порции адреналина, у них нет особых оснований, чтобы не быть осторожными.

В сравнении с оболочками командной строки Unix, которые, в сущности, лишь создают условия для вызова других программ, на Perl легко писать защищенные программы, поскольку он прост и самодостаточен. В отличие от большинства языков программирования оболочек, основанных на многократных проходах каждой строки сценария с таинственными подстановками, Perl использует более общепринятую схему вычислений с меньшим количеством тайных подвохов. Кроме того, поскольку в этом языке больше встроенных функций, он меньше полагается на внешние (и, возможно, не заслуживающие доверия) программы для выполнения своих задач.

В Unix, откуда Perl родом, предпочтительным способом компрометации системы безопасности было уговорить привилегированную программу сделать что-то, для чего она не предназначалась. Чтобы предотвратить такие атаки, Perl разработал уникальный подход к тому, как справиться с враждебной

средой. Perl автоматически включает режим проверки данных на чистоту, когда обнаруживает, что его программа работает с различающимися текущим (effective) и реальным (real) ID пользователя или группы.[1] Даже если у файла, содержащего сценарий Perl, не установлены маски setuid и setgid, все же может оказаться, что этот сценарий выполняется в режиме проверки на чистоту данных. Это произойдет, если ваш сценарий был вызван другой программой, которая *сама* выполнялась с различающимися ID. Программы Perl, которые не были предназначены для работы в режиме проверки меченых данных, могут досрочно прекратить свое существование, если будут захвачены в момент нарушения правил безопасности в режиме проверки чистоты данных. И это правильно, потому что это как раз те интриги, которые исторически затевались в сценариях оболочки для компрометации защиты системы. Perl не так доверчив.

Режим проверки чистоты данных можно активировать и явно с помощью ключа командной строки `-T`. Это нужно делать для демонов, серверов и любых программ, работающих от чужого имени, например сценариев CGI. Программы, которые можно запускать удаленно и анонимно с любой машины в Сети, выполняются в самом враждебном из возможных окружений. Не нужно бояться иногда говорить: «Нет!» В противоположность распространенному мнению можно проявлять значительную осторожность и не стать при этом усохшим морщинистым ханжой.

Для сайтов, заботящихся о безопасности, выполнение всех сценариев CGI с флагом `-T` не просто хороший тон: это закон. Мы не утверждаем, что выполнение в режиме проверки данных на чистоту сделает ваш сценарий безопасным. Не сделает, а простое перечисление того, что сделает, займет целую книгу. Но если вы не выполняете свои сценарии CGI в режиме проверки меченых данных, значит, без всякой необходимости пренебрегли самой сильной защитой, которую может предоставить Perl.

Находясь в этом режиме, Perl предпринимает особые меры предосторожности, чтобы избежать очевидных и скрытых ловушек. Некоторые из этих проверок довольно просты, – например, что не установлены опасные переменные окружения, а каталоги, указанные в переменной PATH, недоступны посторонним для записи; но осторожные программисты всегда использовали такие проверки. Есть, однако, другие проверки, которые лучше всего поддерживаются самим языком, и именно они делают привилегированную программу Perl более надежной, чем соответствующая программа C, а сценарий Perl CGI более надежным, чем написанный на языке без проверки

[1] Бит setuid в правах доступа Unix имеет значение 04000, а бит setgid имеет значение 02000; можно установить один из них или оба, чтобы предоставить пользователю программы некоторые права владельца (или владельцев) программы. (В совокупности их называют программами set-id.) Другие операционные системы могут давать программам особые права другими способами, но принцип остается тем же.

меченых данных. (Которым, насколько нам известно, является любой язык, кроме Perl.)

Принцип прост: нельзя использовать данные, полученные из-за пределов выполняемой программы, для воздействия на что-либо еще, находящееся за ее пределами, во всяком случае намеренно. Любые данные, поступающие в программу извне, становятся «помеченными» как данные сомнительного происхождения (tainted), в том числе все аргументы командной строки, переменные окружения и ввод из файлов. Меченые данные нельзя использовать прямо или косвенно в любой операции, которая вызывает «подоболочку», а также в любых операциях, модифицирующих файлы, каталоги или процессы. Любая переменная, устанавливаемая в выражении, которое ранее ссылалось на меченое значение, сама становится меченой, даже если логически невозможно, чтобы «помеченное» значение повлияло на переменную. Так как «помеченность» ассоциируется с отдельным скаляром, в массиве или хеше одни значения могут быть мечеными, а другие – нет. (В хеше мечеными могут быть только значения, но не ключи.)

В следующем коде иллюстрируется, как будут работать «помеченные» данные, если выполнить по порядку все эти команды. Команды, помеченные словом «ненадежна» создают исключительную ситуацию, а те, которые помечены «OK», – нет.

```
$arg = shift(@ARGV);          # $arg теперь помечена (из-за @ARGV).
$hid = "$arg, 'bar'";         # $hid тоже помечена (из-за  $arg).
$line = <>;                   # Помечена (чтение из внешнего файла).
$path = $ENV{PATH};           # Помечена из-за %ENV, но смотрите ниже.
$mine = 'abc';                # Не помечена.

system "echo $mine";          # Ненадежна, если не установлена PATH.
system "echo $arg";           # Ненадежна: использует sh с меченой $arg.
system "echo", $arg;          # OK, если PATH установлена (не использует sh).
system "echo $hid";           # Ненадежна по двум причинам: помеченная, PATH.

$oldpath = $ENV{PATH};        # $oldpath помеченная(из-за %ENV).
$ENV{PATH} = '/bin:/usr/bin'; # (Разрешает выполнять другие программы.)
$newpath = $ENV{PATH};        # $newpath НЕ помечена.

delete @ENV{qw{IFS
               CDPATH
               ENV
               BASH_ENV}};    # Делает %ENV более безопасной.

system "echo $mine";          # OK, безопасна после переустановки маршрута.
system "echo $hid";           # Ненадежна из-за $hid.

open(OOF, "< $arg");          # OK (open только для чтения не проверяются).
open(OOF, "> $arg");         # Ненадежна  (попытка записи в "помеченный" аргумент).

open(OOF, "echo $arg|")       # Ненадежна из-за помеченной $arg, но...
    or die "невозможность конвейера из echo: $!";

open(OOF,"-|")                # Считается OK: см. ниже "помеченное"
```

```
    or exec "echo", $arg      #   исключение при выполнении списка.
    or die "невозможно exec echo: $!";

open(OOF,"-|", "echo", $arg)   # То же, что предыдущее, OK.
    or die " невозможность конвейера из echo: $!";

$shout = `echo $arg`;         # Ненадежна из-за "помеченности" $arg.
$shout = `echo abc`;          # $shout помечена из-за обратных штрихов.
$shout2 = `echo $shout`;      # Ненадежна из-за "помеченности" $shout.

unlink $mine, $arg;           # Ненадежна из-за "помеченности" $arg.
umask $arg;                   # Ненадежна из-за "помеченности" $arg.

exec "echo $arg";             # Ненадежна из-за "помеченности" $arg, передаваемой
                                оболочке.
exec "echo", $arg;            # Считается OK! (Но смотри ниже.)
exec "sh", '-c', $arg;        # Считается  OK, но в действительности не так!
```

Если вы попытаетесь сделать что-то небезопасное, возникнет исключительная ситуация (которая, не будучи перехвачена, становится фатальной), например, «`Insecure dependency`» или «`Insecure $ENV{PATH}`». См. далее раздел «Наведение порядка в окружении».

Если вы передаете список аргументов в `system`, `exec` или `open` для канала, то происхождение аргументов не проверяется, потому что когда аргументы представлены списком, Perl не требуется вызывать потенциально опасную оболочку, чтобы выполнить команду. Можно с легкостью написать ненадежные `system`, `exec` или `open` для канала с использованием спискового формата, как показано в только что рассмотренном примере. Эти формы освобождаются от проверки, т. к. предполагается, что вы понимаете, что делаете, когда используете их.

Иногда, впрочем, невозможно определить, сколько аргументов передается. Если вы передаете этим функциям массив,[1] который содержит только один элемент, то это все равно что передать одну строку, и при этом может быть использована оболочка. Выход в том, чтобы передать явный путь через косвенные аргументы:

```
system @args;                 # Оболочка вызывается, только когда @args == 1.
system { $args[0] } @args;    # Обходит оболочку, даже если в списке один аргумент.
```

Обнаружение и очистка меченых данных

Чтобы проверить, содержит ли скалярная переменная данные сомнительного происхождения, можно обратиться к функции `is_tainted`, описываемой ниже. Она использует то обстоятельство, что `eval` *`STRING`* создает исключительную ситуацию при попытке компиляции данных сомнительного происхождения. Неважно, что переменная `$nada`, используемая в компилируемом выражении, всегда будет пустой; она все равно окажется меченой, если «помечена» `$arg`. Внешняя проверка `eval` *`BLOCK`* не производит компиляции.

[1] Или функцию, создающую список.

Ее задача – перехватить исключительную ситуацию, создающуюся, когда внутренняя eval получает меченые данные. Поскольке гарантируется, что переменная $@ не будет пустой после каждой eval, если произошла исключительная ситуация, и будет пустой в противном случае, мы возвращаем результат проверки того, что длина содержимого этой переменной равна нулю:

```
sub is_tainted {
    my $arg = shift;
    my $nada = substr($arg, 0, 0);  # нулевая длина
    local $@;  # сохранить версию вызывающей программы
    eval { eval "# $nada" };
    return length($@) != 0;
}
```

Но проверка происхождения данных – лишь часть дела. Как правило, мы прекрасно знаем, какие переменные содержат меченые данные, и надо лишь сбросить признак «помеченности». Единственный официальный путь обхода механизма проверки меченых данных – это ссылка на вложенные соответствия, возвращенные ранее проведенным поиском регулярного выражения.[1] Если ваш шаблон содержит охватывающие скобки, то к захваченным подстрокам можно обращаться через переменные $1, $2 и т. д. либо при вычислении шаблона в списковом контексте. В любом случае предполагается, что вы знали что делали, когда писали шаблон, и написали его для того, чтобы вычистить все, представляющее опасность. Поэтому нужно крепко над ним подумать: не снимайте бездумно «помеченность», иначе вы разрушите весь ее механизм.

Лучше проверять, что переменная содержит только «хорошие» символы, чем проверять, не содержит ли она «плохие», потому что эти последние слишком легко пропустить, если вы о них не подумали. Вот, например, тест, проверяющий, что $string не содержит ничего, кроме «символов слов» (букв, цифр и подчеркиваний), дефисов, знаков @ и точек:

```
if ($string =~ /^([-\@\w.]+)$/) {
    $string = $1;                       # $string теперь не помечена.
}
else {
    die "Плохие данные в $string";         # Записать куда-то в журнал.
}
```

[1] Неофициальный путь состоит в том, чтобы сохранить меченую строку как ключ хеша и получить этот ключ обратно. Поскольку ключи не являются в действительности полноценными SV (внутреннее имя для scalar values – скалярных значений), у них отсутствует свойство «помеченности». Однако такое поведение может в один прекрасный день поменяться, поэтому не следует на него полагаться. Будьте осторожны при работе с ключами, чтобы не снять непреднамеренно «помеченность» со своих данных и не сделать с ними что-нибудь небезопасное.

В результате $string становится достаточно безопасной для дальнейшего использования во внешней команде, т. к. /\w+/ обычно не соответствует метасимволам оболочки, да и другие символы не должны означать для нее чего-либо особенного.[1] Если бы мы использовали /(.+)/s, это было бы небезопасно, поскольку такой шаблон пропускает все. Но Perl это не проверяет. При снятии «помеченности» будьте предельно осторожны со своими шаблонами. Очистка данных с помощью регулярных выражений является *единственным* одобряемым механизмом для снятия «пометки» с «грязных» данных. А иногда это оказывается совершенно неверным подходом. Если вы оказываетесь в режиме проверки меченых данных, т. к. выполняете set-id, а не потому, что специально указали ключ -T, то можете уменьшить риск, ответвив дочерний процесс (forking a child) с меньшими привилегиями; см. раздел «Приведение в порядок окружения».

Прагма use re 'taint' запрещает неявное снятие помеченности при поиске по шаблону до конца текущей лексической области видимости. Эту прагму можно использовать, когда требуется извлечь какие-то подстроки из потенциально меченых данных, но, заботясь о безопасности, лучше оставить эти подстроки мечеными, чтобы уберечься от возможных неприятностей.

Предположим, что вы осуществляете такой поиск, причем $fullpath является меченой:

```
($dir, $file) = $fullpath =~ m!(.*/)(.*)!s;
```

По умолчанию $dir и $file теперь не будут помечены. Но, вероятно, вы не стали бы делать это столь бесцеремонно, если бы задумались о проблемах безопасности. Например, было бы не слишком радостно получить в $file строку "; rm -rf * ;", что можно привести в качестве довольно грубого примера. В следующем примере две результирующие переменные остаются помеченными, если помечена $fullpath:

```
{
    use re 'taint';
    ($dir, $file) = $fullpath =~ m!(.*/)(.*)!s;
}
```

Хорошей стратегией будет по умолчанию оставлять вложенные соответствия мечеными во всем исходном файле и лишь при необходимости избирательно разрешать снятие помеченности во вложенных областях видимости:

```
use re 'taint';
# в остальной части файла $1 и т. д. остаются помеченными
```

---

[1] Если только вы не используете специально искаженные национальные настройки. Perl допускает возможность компрометации национальных настроек системы. Поэтому при выполнении с директивой use locale шаблоны, содержащие символический класс знаков, такие как \w или [[:alpha:]], дают недостоверные результаты.

```
{
    no re 'taint';
    # в этом блоке снимается помеченность при поиске регулярных выражений
    if ($num =~ /^(\d+)$/) {
        $num = $1;
    }
}
```

Ввод из указателя файла или каталога автоматически помечается, за исключением того случая, когда он производится из особого указателя DATA. При желании можно пометить другие указатели как заслуживающие доверия источники с помощью функции untaint из модуля IO::Handle:

```
use IO::Handle;

IO::Handle::untaint(*SOME_FH);          # Либо как процедуру,
SOME_FH->untaint();                     # либо в стиле ООП.
```

Отключение пометки данных для указателя в целом является рискованным шагом. Откуда вы можете знать, что он действительно безопасен? Если вы собираетесь это сделать, то должны хотя бы проверить, что никто, кроме владельца, не может осуществлять запись в этот файл.[1] Если вы работаете с файловой системой Unix (в которой вызов *chown*(2) благоразумно разрешен только суперпользователю), то действителен такой код:

```
use File::stat;
use Symbol 'qualify_to_ref';
sub handle_looks_safe(*) {
    my $fh = qualify_to_ref(shift, caller);
    my $info = stat($fh);
    return unless $info;

    # владельцем не являются ни суперпользователь, ни "я", чей
    # реальный uid находится в переменной $<
    if ($info->uid != 0 && $info->uid != $<) {
        return 0;
    }

    # проверить, могут ли писать в файл группа или другие.
    # используйте также 066 для определения возможности чтения
    if ($info->mode & 022) {
        return 0;
    }
    return 1;
}

use IO::Handle;
SOME_FH->untaint() if handle_looks_safe(*SOME_FH);
```

[1] Хотя можно снимать помеченность и с указателя каталога, эта функция работает только с указателем файла. Это связано с тем, что если передать указатель каталога, то не существует переносимого способа передать его дескриптор файла в stat.

Мы вызывали stat с указателем файла, а не его именем, чтобы избежать опасной ситуации, в которой велика вероятность взаимоблокировок. См. раздел «Обработка ситуаций гонок» далее в этой главе.

Заметим, что эта программа служит лишь хорошим началом. Несколько более параноидальная версия проверила бы еще все родительские каталоги, даже несмотря на то, что нельзя надежно применять stat к указателям каталогов. Но если в какой-нибудь родительский каталог может осуществлять запись кто угодно, знайте, что вы в опасности независимо от наличия условий гонок (race conditions).

У Perl есть свое представление о том, какие операции являются опасными, но можно попасть в беду и с другими операциями, если не проверять, используют ли они меченые значения. Не всегда достаточно лишь проверять входные данные. Функции вывода Perl не проверяют, являются ли мечеными их аргументы, но в некоторых средах это существенно. Если вы не следите за тем, что выводите, то это может привести к положению, когда вы будете выплевывать строки, значение которых окажется неожиданным для того, кто обрабатывает вашу выдачу. Если вы работаете с терминалом, то некоторые управляющие символы и коды могут вызвать необычное для зрителя поведение терминала. Если вы находитесь в среде web и слепо выплевываете обратно переданные вам данные, то можете неведомо для себя создать теги HTML, которые совершенно изменят внешний вид страницы. Что еще хуже, некоторые теги могут даже привести к выполнению кода в броузере.

Представьте себе распространенный случай гостевой книги, в которой пользователи оставляют свои сообщения, которые могут показываться другим посетителям. Гость-злоумышленник может передать специальные теги HTML или поместить в последовательности <SCRIPT>...</SCRIPT>, выполняющие код (например, JavaScript), который отобразится в броузерах последующих посетителей.

Точно так же, как вы должны тщательно проверять наличие только «добропорядочных» символов при просмотре меченых данных, обращающихся к ресурсам вашей собственной системы, следует проявлять осторожность в web-окружении при выводе данных, полученных от пользователя. Например, чтобы удалить все символы, не входящие в список одобренных, попробуйте что-то вроде следующего:

```
$new_guestbook_entry =~ tr[_a-zA-Z0-9 ,./!?()@+*-][]dc;
```

Конечно, не следует использовать это для обработки имени файла, поскольку имена файлов с пробелами и косыми чертами вам едва ли нужны. Но для того чтобы в вашу гостевую книгу не прокрались теги и объекты HTML, этого достаточно. В каждом случае очистки данных есть свои особенности, поэтому всегда потратьте немного времени и решите, что можно и чего нельзя. Механизм проверки меченых данных служит для перехвата глупых ошибок и не отменяет необходимости думать.

Зачистка окружения

Когда вы выполняете из своего сценария Perl другую программу – неважно, каким образом, – Perl проверяет, является ли безопасной ваша переменная PATH. Поскольку она пришла из вашего окружения, ваша переменная PATH сначала помечена, поэтому при попытке запустить другую программу Perl создает исключительную ситуацию «Insecure $ENV{PATH}». Если установить PATH равной известному немеченому значению, Perl удостоверяется, что все каталоги в этом пути недоступны для записи всем, кроме владельца и группы каталога; в противном случае, он создает исключительную ситуацию «Insecure directory».

Вас может удивить, что Perl интересуется вашей PATH даже тогда, когда вы задаете полное имя пути для команды, которую вы хотите выполнить. Верно, если указано абсолютное имя, то PATH не используется для поиска исполняемого модуля, который нужно запустить. Но вы не можете быть уверены в том, что программа, которую вы запускаете, не развернется на 180 градусов и не запустит *другую* программу, с которой возникнут неприятности из-за незащищенной PATH. Поэтому Perl требует, чтобы вы установили надежное значение PATH, прежде чем вызывать какую-либо программу каким бы то ни было способом.

PATH не является единственной переменной, которая может принести огорчения. Поскольку некоторые оболочки используют переменные IFS, CDPATH, ENV и BASH_ENV, Perl проверяет, чтобы все они были либо пусты, либо непомечены перед тем, как запустить другую команду. Либо присвойте этим переменным безопасные значения, либо вообще удалите из окружения:

```
delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};   # Сделайте %ENV более безопасным
```

Возможности системы, предоставляющие удобство в обычной среде, могут стать проблемой для защиты данных во враждебной среде. Даже если вы не забываете запрещать имена файлов, содержащие символы перевода строки, важно помнить, что open обращается не только к именованным файлам. Если надлежащим образом украсить аргумент, представляющий имя файла, то вызовы open с одним или двумя аргументами могут запускать произвольные внешние программы через каналы, ответвлять дополнительные экземпляры текущего процесса, дублировать указатели файлов и интерпретировать специальное имя файла «-» как псевдоним для стандартного ввода или вывода. Они могут также игнорировать ведущие и замыкающие пробельные символы, что скроет такие необычные аргументы от ваших проверочных шаблонов. Хотя верно, что проверка меченых данных в Perl перехватывает меченые аргументы, используемые при открытии конвейера (если только не используется список разделяемых аргументов), и любое открытие файла, кроме как только для чтения, но создаваемая при этом исключительная ситуация может нарушить нормальное выполнение вашей программы.

Если вы собираетесь использовать полученные извне данные в качестве имени открываемого файла, хотя бы включите явно через пробел режим откры-

тия. Однако безопаснее всего, по-видимому, использовать низкоуровневую sysopen или формат open с тремя аргументами:

```
# Волшебное открытие - может быть любым
open(FH, $file)             or die "невозможно волшебно открыть $file: $!";

# или fork, но все же принимает дескрипторы файлов и "-",
# и игнорирует пробельные символы с обоих концов имени.
open(FH, "< $file")         or die "невозможно открыть $file: $!";

# WYSIWYG-открытие: отключает все удобные функции.
open(FH, "<", $file)        or die "невозможно открыть $file: $!";

# Те же свойства, что в 3-аргументной версии WYSIWYG.
require Fcntl;
sysopen(FH, $file, O_RDONLY)        or die "невозможно sysopen $file: $!";
```

Даже этих шагов не вполне достаточно. Perl не препятствует открытию для чтения файлов с мечеными именами, поэтому нужно быть осторожным в отношении того, что вы показываете людям. Программа, которая открывает файл с произвольным именем, указанным пользователем, и показывает его содержимое, остается проблемой для защиты данных. А если это личное письмо? А если это системный файл с паролями? А если это данные о зарплате или вашем портфеле ценных бумаг?

Взгляните пристально на имена файлов, полученные от потенциально враждебного пользователя,[1] прежде чем открывать их. Например, можно проверить, чтобы в пути не было подлых составляющих каталогов. Известными трюками такого сорта являются имена типа «../../../../../../../etc/passwd». Защититься можно, проверив, чтобы в имени пути не было косой черты (если она служит разделителем каталогов в вашей системе). Другой часто используемый прием состоит в помещении символов перевода строки или точки с запятой в имя файла, в результате чего впоследствии какой-нибудь несчастный глупый интерпретатор командной строки поймет их как начало новой команды посреди имени файла. Вот поэтому в режиме проверки меченых данных отвергаются непроверенные внешние команды.

Доступ к командам и файлам при ограниченных правах

Последующее изложение касается некоторых ловких приемов, относящихся к безопасности на Unix-подобных системах. Пользователи других систем могут спокойно (или, скорее, не спокойно) пропустить этот раздел.

Если вы выполняете программу set-id, постарайтесь по возможности выполнять опасные операции с правами пользователя, а не программы. Это означает, что, собираясь вызвать open, sysopen, system, обратный штрих или

[1] А в Сети единственными пользователями, которых вы можете не считать потенциально враждебными, являются *активно* враждебные пользователи.

другие операции с файлами или процессами, вы можете защитить себя, установив текущие UID или GID снова в реальные UID или GID. В Perl можно сделать это для сценариев setuid, сказав `$> = $<` (или `$EUID = $UID` при `use English`), а для сценариев setgid, сказав `$( = $)` (`$EGID = $GID`). Если установлены оба ID, нужно переустановить оба. Однако иногда это недостижимо, поскольку расширенные права доступа могут все же понадобиться позже в вашей программе.

В таких случаях Perl предоставляет достаточно надежный способ открытия файла или канала из программы set-id. Во-первых, нужно ответвить дочерний процесс с помощью особого синтаксиса `open`, связывающего родительский и дочерний процессы через канал. В дочернем процессе установите ID пользователя и группы в исходные или заведомо безопасные значения. Вы сможете также изменить атрибуты дочернего процесса, не затрагивая родительский, что позволит изменить рабочий каталог, установить маску создания файла и переменные окружения. Выполняясь без дополнительных прав доступа, дочерний процесс, наконец, вызывает `open` и передает любые данные, к которым ему удается получить доступ, от имени простого, но слабоумного пользователя своему могущественному, но поистине параноидальному родителю.

В то время как `system` и `exec` не используют оболочку, если вы передаете им более одного аргумента, оператор «обратный штрих» не имеет такого альтернативного соглашения по вызову. С помощью приема ветвления мы легко эмулируем обратный штрих, не опасаясь интерпретации его оболочкой и с уменьшенными (а потому более безопасными) правами доступа:

```
use English;   # чтобы использовать $UID и прочие
die "Невозможно породить через open: $!"   unless defined($pid = open(FROMKID, "-
|"));
if ($pid) {           # родитель
    while (<FROMKID>) {
        # какие-то действия
    }
    close FROMKID;
}
else {
    $EUID = $UID;  # setuid(getuid())
    $EGID = $GID;  # setgid(getgid()), и initgroups(2) по getgroups(2)
    chdir("/")      or die "невозможно chdir в /: $!";
    umask(077);
    $ENV{PATH} = "/bin:/usr/bin";
    exec 'myprog', 'arg1', 'arg2';
    die "невозможно exec myprog: $!";
}
```

Это гораздо лучший способ вызова других программ из сценария set-id. Гарантируется, что вы не будете использовать оболочку для выполнения чего-либо и что вы потеряете свои права перед тем, как самому выполнить программу. (Но из-за того, что списковые форматы `system`, `exec` и открытия

канала не подвергаются проверке чистоты своих аргументов, вы все же должны быть осторожны в отношении того, что вы передаете.)

Если вам не нужно терять права доступа и требуется реализовать обратный штрих или открытие канала, не подвергаясь риску перехвата оболочкой ваших аргументов, можно использовать такой код:

```
open(FROMKID, "-|") or exec("myprog", "arg1", "arg2")
    or die "can't run myprog: $!";
```

а затем просто читать из FROMKID в родительском процессе. В версии Perl 5.6.1 можно записать это так:

```
open(FROMKID, "-|", "myprog", "arg1", "arg2");
```

Польза технологии ветвления не только в запуске команд из программы setid. Она удобна и для открытия файлов с ID того, кто запустил программу. Допустим, что у вас есть программа setuid, которой нужно открыть файл для записи. Вы не хотите выполнять open со своими дополнительными правами доступа, но окончательно потерять их вы тоже не можете. В таком случае сделайте ответившуюся копию, которая отменяет свои дополнительные права, чтобы сделать для вас open. Если вы хотите осуществить запись в файл, сделайте запись в дочерний процесс, а уж он сделает запись в файл вместо вас.

```
use English;

defined ($pid = open(SAFE_WRITER, "|-"))
    or die "Невозможно ветвление: $!";

if ($pid) {
    # вы родитель. выводите данные в дочерний SAFE_WRITER
    print SAFE_WRITER "@output_data\n";
    close SAFE_WRITER
        or die $! ? "Syserr при закрытии SAFE_WRITER: $!"
                  : "Статус ожидания $? от SAFE_WRITER ";
}
else {
    # вы дочерний процесс, так что отмените дополнительные права
    ($EUID, $EGID) = ($UID, $GID);

    # открыит файл с правами исходного пользователя
    open(FH, "> /some/file/path")
        or die "невозможно открыть /some/file/path для записи: $!";

    # копировать от родителя (теперь это stdin) в файл
    while (<STDIN>) {
        print FH $_;
    }
    close(FH)   or die "ошибка при закрытии: $!";
    exit;       # Не забудьте сделать так, чтобы SAFE_WRITER исчез.
}
```

При невозможности открыть файл дочерний процесс выводит сообщение об ошибке и завершает работу. Если родительский процесс станет выводить данные по не действующему теперь указателю файла в дочернем процессе, вырабатывается сигнал разрушения канала (SIGPIPE), являющийся фатальным, если он не перехватывается или игнорируется. См. раздел «Сигналы» в главе 16 «Межпроцессное взаимодействие».

Обработка ошибок синхронизации

Иногда работа вашей программы очень чувствительна к временным моментам внешних событий, находящихся вне вашей власти. Это всегда становится проблемой, когда другие программы, в особенности враждебные, соперничают с вашей программой за одни и те же ресурсы (например, файлы или устройства). В многозадачной среде невозможно предсказать порядок, в котором процессы, ожидающие своей очереди выполнения, получат доступ к процессору. Потоки команд возможных процессов чередуются, поэтому сначала один процесс получает на какое-то время CPU, затем другой процесс и т. д. Очередность выполнения и отводимое на него время кажутся случайными. Если программа одна, то проблем не возникает, в отличие от ситуации, когда несколько программ совместно используют общие ресурсы.

Особенно чувствительны к таким вопросам программисты, разрабатывающие потоки. Они быстро усваивают, что нельзя говорить:

```
$var++ if $var == 0;
```

когда нужно сказать:

```
{
    lock($var);
    $var++ if $var == 0;
}
```

В первом случае получаются непредсказуемые результаты, если несколько потоков пытаются одновременно выполнять этот код. (См. главу 17 «Потоки».) Если совместно используемыми объектами являются файлы, а процессы как потоки соперничают за доступ к ним, возникают такие же проблемы. Процесс, в конце концов, это, в некотором отношении, лишь поток. Или наоборот.

Непредсказуемость во времени оказывает воздействие как на те ситуации, в которых затрагиваются права доступа, так и на обычные. Сначала мы опишем, как справиться с давней ошибкой в старых ядрах Unix, проявляющейся в программах set-id. Затем мы перейдем к обсуждению условий гонок (race conditions) в целом, возможности возникновения из-за них брешей в защите и действий, помогающих этого избежать.

Ошибки защиты в ядре Unix

Помимо тех очевидных проблем, которые проистекают из того, что особые права даются таким гибким и непроницаемым интерпретаторам, какими являются оболочки, в старых версиях Unix в ядре имеется ошибка, которая делает небезопасными все сценарии set-id даже раньше, чем они попадут интерпретатору. Проблема состоит не в самом сценарии, а в ситуации, возникающей при обнаружении ядром сценария set-id для исполнения. (Этой ошибки нет в системах, ядро которых не распознает `#!`.) Когда ядро открывает такой файл, чтобы узнать, какой интерпретатор нужно запустить, возникает некоторая задержка, прежде чем запустится интерпретатор (теперь set-id) и откроет этот файл. Эта задержка дает возможность злоумышленнику изменить файл, особенно если система поддерживает символические ссылки.

К счастью, иногда эту «возможность» ядра можно отключить. К несчастью, сделать это можно разными способами. Система может запретить сценарии с установленными битами set-id, что не очень удобно. Кроме того, она может игнорировать биты set-id у сценариев. В последнем случае Perl может эмулировать механизм setuid и setgid, обнаружив биты set-id (в другом отношении бесполезные) у сценариев Perl. Делает он это с помощью особого исполняемого модуля *suidperl*, который при необходимости автоматически вызывается.[1]

Однако если в ядре не отключена функция сценариев set-id, то Perl начинает громко жаловаться, что данный сценарий setuid небезопасен. Придется либо отключить для ядра «возможность» сценариев set-id, либо поместить сценарий в C-оболочку. C-оболочка – это просто скомпилированная программа, единственным действием которой является вызов вашей программы на Perl. Скомпилированные программы не подвержены действию той ошибки ядра, от которой страдают сценарии set-id.

Вот простая оболочка, написанная на C:

```
#define REAL_FILE "/path/to/script"
main(ac, av)
    char **av;
{
    execv(REAL_FILE, av);
}
```

Скомпилируйте ее в исполняемый образ и запускайте вместо своего сценария set-id. Используйте абсолютное имя файла, т. к. C недостаточно сообразителен, чтобы проверять меченые данные по вашей переменной `PATH`.

---

[1] При необходимости и разрешении на это. Если Perl обнаружит, что файловая система, в которой находится сценарий, смонтирована с параметром `nosuid`, то это значение будет соблюдаться. Здесь нельзя использовать Perl, чтобы обойти политику защиты данных вашего системного администратора.

(Другой возможный подход состоит в применении экспериментального генератора кода C в компиляторе Perl. В скомпилированном образе вашего сценария не будет ситуации гонок. См. главу 18 «Компиляция».)

В последние годы стали, наконец, поставляться системы с исправленной ошибкой set-id. В этих системах при передаче интерпретатору имени сценария set-id больше не используется имя файла, которое можно подделать, а передается особый файл, представляющий дескриптор файла, например */dev/fd/3*. Этот особый файл уже открыт для сценария, поэтому не может возникнуть ситуации гонки, которую используют злонамеренные сценарии.[1] Самые последние версии Unix используют такой подход, чтобы избежать ситуации гонки, возникающей при открытии одного и того же имени файла дважды.

Обработка ситуаций гонки

Вот нам и приходится разбираться с ситуациями гонок (race conditions). Чем же они являются в действительности? Они часто всплывают при обсуждении проблем защиты данных. (Хотя реже, чем они возникают в небезопасных программах. К несчастью.) Дело в том, что они служат обильным источником тонких программных ошибок, а такие ошибки часто становятся основой *подвигов* на ниве защиты данных (если вежливо назвать таким словом взлом системы защиты данных). Возникает обсуждаемая ситуация, когда результат нескольких взаимосвязанных событий зависит от порядка возникновения этих событий, но этот порядок нельзя гарантировать из-за недетерминированных временных эффектов. Каждое событие стремится осуществиться первым, а о конечном состоянии системы можно только гадать.

Представьте себе, что у вас есть один процесс, который перезаписывает существующий файл, и другой процесс, который читает тот же самый файл. Нельзя предсказать, что вы прочтете: старый файл, новый файл или случайную смесь их обоих. Неизвестно даже, все ли данные прочтены. Читающая программа может *выиграть гонку*, прочесть файл до конца и завершить работу. Тем временем пишущая программа продолжает свою работу уже после того, как читающая программа нашла конец файла, файл продолжится далее того места, где было прекращено его чтение, а читающая программа об этом и не узнает.

В данном случае решение простое: обе стороны должны блокировать файл с помощью `flock`. Обычно читающая программа запрашивает блокировку с совместным доступом, а пишущая программа – с монопольным доступом. Если все стороны запрашивают эти рекомендательные блокировки и соблюдают их, то чтение и запись не могут перемежаться и искалеченных данных быть не может. См. раздел «Блокировка файлов» главы 16.

---

[1] В этих системах нужно компилировать Perl с ключом `-DSETUID_SCRIPTS_ARE_SECURE_NOW`. Программа *Configure*, которая собирает Perl, сама старается это определить, чтобы вам не пришлось задавать это явным образом.

Значительно менее очевидная форма ситуации гонки может возникнуть, когда операции над именем файла управляют последующими операциями над этим файлом. Когда операторы проверки файлов осуществляются с именами файлов, а не с их указателями, это прямая дорога к возникновению ситуации гонок. Рассмотрим такой код:

```
if (-e $file) {
    open(FH, "< $file")
        or die "невозможно открыть $file для чтения: $!";
}
else {
    open(FH, "> $file")
        or die "невозможно открыть $file для записи: $!";
}
```

Этот код кажется совершенно простым и, тем не менее, может вызвать ситуацию гонок. Нет никакой гарантии, что результат выполнения проверки -e все еще будет действителен при вызове того или другого open. В блоке if другой процесс может удалить файл, прежде чем удастся его открыть, таким образом, файл, который, как вам казалось, должен быть на месте, найден не будет. В блоке else другой процесс может создать файл, прежде чем второй open получит возможность создать файл, и файл, которого, как вам казалось, нет, окажется там. Простая функция open создает новые файлы, но перезаписывает существующие. Допустим, мы решили перезаписать любой существующий файл, но что если этот файл является вновь созданным псевдонимом или символической ссылкой на другой файл в системе, который нам очень не хотелось бы перезаписать? Можно считать, что значение каждого имени файла в каждый данный момент известно, но в действительности в этом никогда нельзя быть уверенным, если в системе выполняются другие процессы с правом доступа к каталогу, содержащему этот файл.

Чтобы решить описанную проблему с перезаписью, нужно использовать функцию sysopen, которая позволяет отдельно управлять тем, нужно ли создавать новый файл и можно ли разрушить существующий. И мы откажемся от проверки существования файла -e, поскольку она не приносит здесь никакой пользы, а только увеличивает шансы попасть в ситуацию гонок.

```
use Fcntl qw/O_WRONLY O_CREAT O_EXCL/;
open(FH, "<", $file)
    or sysopen(FH, $file, O_WRONLY | O_CREAT | O_EXCL)
    or die "невозможно создать новый файл $file: $!";
```

Теперь, даже если файл каким-то образом возникнет в промежутке времени между отказом open и попыткой sysopen открыть новый файл для записи, никакого вреда не будет, потому что с теми флагами, которые указаны, sysopen не станет открывать уже существующий файл.

Если кто-то попытается заставить вашу программу дурно себя вести, весьма вероятно, что он будет делать это, создавая и уничтожая файлы, когда вы этого не ожидаете. Один из способов снизить риск быть обманутым состоит

в том, чтобы дать себе слово никогда не выполнять операции над файлом с одним и тем же именем больше одного раза. Как только вы открыли файл, забудьте его имя (вспоминая, может быть, только в сообщениях об ошибках) и действуйте только с указателем, представляющим файл. Это значительно безопаснее, т. к. даже если кто-то сможет манипулировать именами ваших файлов, делать это с указателями он не сумеет. (Только если вы дадите ему разрешение – см. раздел «Передача указателей файлов» в главе 16.)

Ранее в этой главе мы продемонстрировали функцию `handle_looks_safe`, которая вызывала функцию Perl `stat` с указателем (не именем) файла для проверки владельца и прав доступа к файлу. Использование указателя файла важно для корректности: если бы мы работали с именем файла, не было бы никакой гарантии, что файл, атрибуты которого исследовались, тот же самый, что и только что открытый (или который собирались открыть). Какой-нибудь злоумышленный негодяй мог удалить наш файл и быстро подменить его своим с какой-нибудь гадостью в некий момент между `stat` и `open`. Неважно, которая из них вызывается первой; возможность нечестной игры в перерыве между ними в любом случае сохраняется. Кто-то может решить, что риск очень мал, т. к. промежуток времени очень мал, но существует множество сценариев взлома, которые будут счастливы запустить вашу программу тысячу раз, чтобы поймать единственный случай, когда она окажется недостаточно осторожной. Толковый сценарий взлома может даже понизить приоритет вашей программы, чтобы она прерывалась чаще, и несколько ускорить события. Люди напряженно работают над такими вещами, – поэтому их и называют *подвигами* (*exploits*).

Вызывая `stat` с указателем файла, который уже открыт, мы используем имя файла только один раз и поэтому избегаем ситуации гонок. Хорошей стратегией для устранения гонок между двумя событиями является попытка объединения их в одно и придание операции атомарного характера.[1] Поскольку мы обращаемся к файлу по имени только один раз, возникновения гонки в промежутках между несколькими обращениями быть не может, поэтому не важно, изменится ли имя. Даже если взломщик удалит открытый нами файл (да, бывает и такое) и поместит вместо него другой, чтобы обмануть нас, мы сохраним указатель на настоящий исходный файл.

Временные файлы

Помимо выходов за пределы буферов (которым сценарии Perl практически не подвержены) и доверия не заслуживающим того входным данным (защитой от чего служит режим проверки чистоты данных), неправильное создание временных файлов является одной из наиболее часто используемых брешей в системе защиты. К счастью, атака на временные файлы обычно

[1] Да, операции над атомами можно проводить и в безъядерной зоне. Когда Демокрит назвал «атомами» невидимые частицы вещества, он буквально имел в виду нечто неделимое: *a* (не) + *tomos* (делимый). Атомарная операция – это действие, которое нельзя прервать. (Попробуйте как-нибудь прервать атомную бомбу.)

требует от взломщиков наличия законной учетной записи пользователя, что существенно сокращает число потенциальных негодяев.

Небрежные и случайные программы используют временные файлы различными небезопасными способами, например, помещая их в доступные кому угодно по записи каталоги, используя легко угадываемые имена и не проверяя существование файла с тем же именем. Если вы обнаружите программу с кодом типа следующего:

```
open(TMP, ">/tmp/foo.$$")
    or die "can't open /tmp/foo.$$: $!";
```

знайте, что вы столкнулись с тремя указанными ошибками сразу. Эта программа просто напрашивается на неприятности.

Способ взлома состоит в том, что взломщик подбрасывает файл с тем же именем, которое вы собираетесь использовать. Включение в него PID не является достаточным для уникальности – как это ни удивительно, угадать PID не очень трудно.[1] После этого появляется программа с неосторожным вызовом `open`, которая вместо создания временного файла для своих личных целей перезаписывает файл взломщика.

Так какой вред может от этого случиться? Разнообразный. Файл взломщика, как можно догадаться, это не простой файл. Это символическая ссылка (а иногда и жесткая ссылка), возможно, указывающая на некоторый важный файл, в который взломщик обычно не может самостоятельно осуществлять запись, например */etc/passwd*. Программа считает, что открыла в */tmp* совершенно новый файл, но на самом деле она разрушила существующий файл в каком-то другом месте.

В Perl есть две функции, которые при правильном применении помогают решить эти проблемы. Первая из них – `POSIX::tmpnam`, которая возвращает имя файла, предполагаемого к самостоятельному открытию:

```
# Продолжать использовать имена, пока не получим совершенно новое.
use POSIX;
do {
    $name = tmpnam();
} until sysopen(TMP, $name, O_RDWR | O_CREAT | O_EXCL, 0600);
# Теперь осуществлять ввод-вывод с помощью указателя TMP.
```

Второй является `IO::File::new_tmpfile`, которая возвращает указатель уже открытого файла:

```
# Или позволить модулю сделать это вместо нас.
use IO::File;
my $fh = IO::File::new_tmpfile();  # this is POSIX's tmpfile(3)
# Теперь осуществлять ввод-вывод с помощью указателя $fh.
```

[1] Если только у вас не система типа OpenBSD, которая присваивает новые PID случайным образом.

Ни тот, ни другой методы не совершенны, но лучшим из двух является первый. Основной недостаток, присущий второму способу, заключается в том, что Perl зависит от слабостей в той реализации *tmpfile*(3), которая будет присутствовать в системной библиотеке C, и вы не можете быть застрахованы от того, что эта функция не делает чего-либо столь же опасного, как `open`, которую мы пытаемся исправить. (А некоторые, к сожалению, делают.) Менее существенная проблема состоит в том, что эта функция вообще не возвращает имя файла. Хотя лучше будет, если вы сможете работать с временным файлом без имени, поскольку в результате вы не спровоцируете ситуацию гонки его повторным открытием, но часто это невозможно.

Первый способ нехорош главным образом тем, что не позволяет управлять маршрутом, как функция *mkstemp*(3) из библиотеки C. Во-первых, всегда нежелательно помещать файл в сетевую файловую систему. Флаг `O_EXCL` не гарантирует правильной работы в сетевой файловой системе, поэтому несколько процессов, запрашивающих монопольное создание примерно в одно время, могут все преуспеть в этом. Во-вторых, поскольку запрашиваемый путь находится, возможно, в каталоге, в который могут осуществлять запись другие, кто-нибудь может подкинуть символическую ссылку, указывающую на несуществующий файл, что заставит вас создать свой файл в том месте, которое выбрано не вами.[1] Если необходимо что-то сохранить во временных файлах, не помещайте их в каталог, право записи в который имеет каждый. Если это необходимо, обеспечьте использование флага `O_EXCL` в `sysopen` и постарайтесь работать с каталогами, для которых установлен флаг «удаляет только владелец» – так называемый «липкий» (sticky) бит.

В версии Perl 5.6.1 появляется третий путь. Стандартный модуль `File::Temp` учитывает все сложности, о которых мы говорили. Можно так использовать опции по умолчанию:

```
use File::Temp "tempfile";
$handle = tempfile();
```

Или задать некоторые опции следующим образом:

```
use File::Temp "tempfile";
($handle, $filename) = tempfile("plughXXXXXX",
                                DIR => "/var/spool/adventure",
                                SUFFIX = '.dat');
```

Модуль `File::Temp` предоставляет также заботящиеся о безопасности эмуляции других упомянутых нами функций (хотя родной интерфейс лучше, поскольку дает нам открытый указатель файла, а не просто имя файла, подверженное ситуации гонки). См. в главе 32 «Стандартные модули» более подробное описание опций и семантики этого модуля.

---

[1] Решением, но только в некоторых операционных системах, является вызов `sysopen` и OR с флагом `O_NOFOLLOW`. Это вызывает неудачу выполнения функции, если конечная составляющая пути является символической ссылкой.

Получив указатель файла, с ним можно делать что угодно. Он открыт для чтения и записи, поэтому можно писать по указателю, перемещаться к началу и при необходимости переписывать то, что выведено. Но чего действительно нужно избегать, так это повторного открытия того же имени файла, потому что нельзя быть уверенным, что это тот же самый файл, который был открыт в первый раз.[1]

Когда вы запускаете из своего сценария другую программу, Perl обычно закрывает все указатели файлов, чтобы прикрыть другую брешь в системе безопасности. Если вы используете `fcntl`, чтобы сбросить флаг закрытия при exec (как показано в конце описания `open` в главе 29 «Функции»), то другие вызываемые программы наследуют этот новый указатель открытого файла. В системах, поддерживающих каталог */dev/fd/*, можно передать другой программе имя файла, в действительности означающее дескриптор файла, построив его так:

```
$virtname = "/dev/fd/" . fileno(TMP);
```

Если необходимо вызвать подпрограмму или программу Perl, которые рассчитывают получить в качестве аргумента имя файла, и вы знаете, что эта подпрограмма или программа вызывают для его открытия обычную функцию `open`, можно передать указатель с использованием обозначений Perl для указателя файла:

```
$virtname = "=&" . fileno(TMP);
```

Когда такое «имя» передается обычной функции `open` с одним или двумя аргументами (но не тремя, которые лишат силы это полезное волшебство), вы получаете доступ к дублированному указателю. В некоторых отношениях такая конструкция более переносима, чем передача файла из */dev/fd/*, поскольку она работает везде, где работает Perl, а не только в системах, в которых есть каталог */dev/fd/*. С другой стороны, такой специальный синтаксис `open` для доступа к дескрипторам файлов по номеру работает только с программами Perl, но не с программами, написанными на других языках.

Работа с ненадежным кодом

Механизм меченых данных – это всего лишь своего рода гарантия безопасности, необходимая для перехвата фальшивых данных, которые вы должны были бы перехватить сами, но не позаботились об этом, а передали в систему. Он несколько напоминает необязательные предупредительные сообщения, которые может выдавать Perl; реальной проблемы при этом может не быть, но в среднем потери от действий при ложных утверждениях меньше потерь

[1] За исключением того, чтобы выполнить `stat` с обоими указателями и сравнить первые два возвращаемые значения каждого (пары device/inode). Но тогда будет уже слишком поздно, потому что повреждения уже нанесены. Все, что можно сделать, – это обнаружить ущерб и произвести аварийный выход (и, может быть, послать письмо системному администратору).

от бездействия при ложных отрицаниях. Что касается меченых данных, то последние потери еще более бросаются в глаза, т. к. фальшивые данные не просто дают неправильный ответ, они могут загубить вам систему и пару предыдущих лет работы. (А может быть, и пару последующих лет, если вы не сделали хороших резервных копий.) Режим проверки чистоты данных полезен, когда вы уверены, что написали честный (правильный) код, но не очень верите тем, кто снабжает вас данными, и допускаете, что они обманным путем могут попытаться вовлечь вас в нечто недопустимое.

Данные – это одна сторона вопроса. Совсем другая сторона проявляется, когда вы не доверяете даже коду, который выполняете. А что если вы загрузите из Сети приложение, которое содержит вирус, или бомбу с часовым механизмом, или троянского коня? Проверка на чистоту данных при этом бесполезна, потому что данные, которые вы загружаете в программу, могут быть прекрасными; не вызывает доверия сам код. Вы ставите себя в такое же положение, как при получении от незнакомого человека неизвестного устройства с запиской, в которой сказано: «Просто приложите это к голове и потяните за крючок». Возможно, вы решите, что это устройство для сушки волос, но вы не долго будете так думать.

В этой области осторожность является синонимом паранойи. Что вам нужно, так это система, позволяющая поместить подозрительный код в карантин. Код продолжает существовать и даже выполнять некоторые функции, но ему не позволяется лазать всюду и делать, что заблагорассудится. В Perl такого рода карантин можно осуществить с помощью модуля `Safe`.

Защищенные разделы

Модуль `Safe` позволяет создать некий особый раздел – *sandbox* (*песочница*), в котором перехватываются все системные операции и тщательно контролируется доступ к пространству имен. Низкоуровневые технические детали этого модуля находятся в процессе непрерывного изменения, поэтому мы предпочли более философский подход.

Ограничение доступа к пространству имен

В самых общих чертах объект `Safe` напоминает сейф, но только идея состоит в том, чтобы нехороших людей впускать и не выпускать. В мире Unix существует системный вызов с именем *chroot*(2), который может установить для процесса постоянное ограничение, заставив его выполняться в некотором подкаталоге структуры каталогов, – создать для него, если хотите, маленький персональный ад. Будучи помещен туда, процесс лишен возможности обращаться к находящимся снаружи файлам, поскольку не имеет способа *назвать* эти файлы.[1] Объект `Safe` делает нечто похожее, только ограни-

[1] На некоторых сайтах это делается для выполнения всех сценариев CGI, с применением «закольцованного» монтирования «только для чтения». Такая настройка требует усилий, но если кто-нибудь когда-нибудь сбежит, то обнаружит, что идти ему некуда.

чивается не подмножеством структуры каталогов файловой системы, а подмножеством структуры пакетов Perl, которая тоже иерархична, как и файловая система.

Можно и по-другому взглянуть на объект Safe – как на специальную комнату с полупрозрачными зеркалами, в которую полиция помещает подозрительных типов. Снаружи в комнату заглянуть можно, а изнутри видеть, что делается снаружи, нельзя.

При создании объекта Safe можно дать ему имя пакета. Если этого не сделать, новое имя будет выбрано за вас:

```
use Safe;
my $sandbox = Safe->new("Dungeon");
$Dungeon::foo = 1;  # Однако прямой доступ не одобряется.
```

Если вы полностью квалифицируете переменные и функции с помощью имени пакета, переданного методу new, то можете обращаться к переменным этого пакета снаружи, по крайней мере, в текущей реализации. Однако это может измениться, т. к. в настоящее время планируется клонировать таблицу символов в новый интерпретатор. Несколько более совместимым вперед может быть выполнение действий до создания Safe, как показано ниже. Такой код, возможно, сохранит работоспособность и представляет собой удобный способ настройки объекта Safe, который должен начинать работу с большим числом «состояний». (Мы согласны, что $Dungeon::foo не является большим числом состояний.)

```
use Safe;
$Dungeon::foo = 1;  #По-прежнему прямой доступ, по-прежнему не одобряется.
my $sandbox = Safe->new("Dungeon");
```

Но Safe предоставляет способ доступа к глобальным переменным раздела, даже если вы не знаете имени его пакета. Поэтому для максимальной совместимости вперед (но не для максимальной скорости) мы рекомендуем метод reval:

```
use Safe;
my $sandbox = Safe->new();
$sandbox->reval('$foo = 1');
```

(На самом деле это тот метод, который применяется для выполнения подозрительного кода.) Код, передаваемый в безопасный раздел для компиляции и выполнения, считает, что он на самом деле находится в пакете main. То, что внешней среде известно как $Dungeon::foo, код внутри рассматривает как $main::foo или $::foo, или просто $foo, если не включена директива use strict. Внутри раздела обращение $Dungeon::foo не будет работать, поскольку на самом деле оно будет обращением к $Dungeon::Dungeon::foo. В результате того, что объект Safe имеет свое представление о main, переменные и подпрограммы в остальной части вашей программы оказываются защищенными.

Чтобы скомпилировать и запустить код внутри раздела, используйте метод reval («restricted eval»), передавая строку кода в качестве аргумента. Как и в любой другой конструкции eval *STRING*, ошибки компиляции и исключительные ситуации времени исполнения в reval не убьют вашу программу. Они просто осуществляют аварийный выход из reval и оставляют исключительную ситуацию в переменной $@, поэтому обязательно проверяйте ее после каждого вызова reval.

С учетом произведенной ранее инициализации этот код выведет «foo теперь 2»:

```
$sandbox->reval('$foo++; print "foo теперь $main::foo\n"');
if ($@) {
    die "Невозможно компилировать код в разделе: $@";
}
```

Для того чтобы только скомпилировать код, но не выполнять его, заключите свою строку в объявление подпрограммы:

```
$sandbox->reval(q{
    our $foo;
    sub say_foo {
        print "foo теперь $main::foo\n";
    }
}, 1);
die if $@;      # проверка компиляции
```

На этот раз мы передали reval второй аргумент, который, будучи истинным, требует от reval компилировать код с директивой strict. Из самой же строки кода отключить strict нельзя, потому что импорт и отмена импорта находятся в числе вещей, которые нельзя обычным образом выполнять в разделе Safe — см. следующий раздел.

Если вы создали в разделе функцию say_foo, то следующие строки действуют примерно одинаково:

```
$sandbox->reval('say_foo()');        # Лучший способ.
die if $@;

$sandbox->varglob('say_foo')->();    # Вызов через анонимную глобальную.

Dungeon::say_foo();                  # Прямой вызов, очень не одобряется.
```

Ограничение доступа к операторам

Другая важная особенность объекта Safe состоит в том, что Perl ограничивает операции, которые можно выполнять в песочнице (sandbox). (Вы можете позволить своему ребенку взять в песочницу ведерко и лопатку, но на базуку, видимо, наложите запрет.) Недостаточно защитить оставшуюся часть программы, нужно защитить и оставшуюся часть компьютера.

При компиляции кода Perl в объекте Safe с помощью reval или rdo (ограниченная версия оператора do *FILE*) компилятор сверяется с особым списком

пораздельного управления доступом и проверяет допустимость компиляции каждой отдельной операции. Благодаря этому не нужно проявлять (особое) беспокойство относительно непредвиденных управляющих символов оболочки, незапланированного открытия файлов, странных утверждений кода в регулярных выражениях и большинства проблем с внешним доступом, по поводу которых обычно волнуются. (Или должны волноваться.)

Интерфейс для указания операторов, которые должны разрешаться или запрещаться, в данное время подвергается переработке, поэтому мы покажем только, как используется набор по умолчанию. За подробностями обратитесь к электронной документации по модулю `Safe`.

Модуль `Safe` не предоставляет полной защиты от атак типа *отказа в обслуживании (DoS attack)*, особенно в случае применения в более разрешительных режимах. Атаки типа «отказ в обслуживании» захватывают все имеющиеся в системе ресурсы некоторого вида, в результате чего другим процессам отказывается в доступе к важным системным средствам. Примерами таких атак служат переполнение таблицы процессов ядра, захват CPU с помощью жесткого бесконечного цикла, израсходование имеющейся памяти и переполнение файловой системы. Такие проблемы очень тяжело решать, особенно переносимым способом. Атаки типа «отказ в обслуживании» рассмотрены в конце раздела «Код, маскирующийся под данные».

Примеры использования модуля Safe

Представьте, что у вас есть программа CGI, управляющая формой, в которую пользователь может ввести произвольное выражение Perl и получить результат его выполнения.[1] Как и любые внешние данные, эта строка будет меченой, поэтому Perl пока не позволит ее выполнить: сначала нужно снять помеченность с помощью поиска по шаблону. Проблема в том, что невозможно разработать шаблон, который выявит любую возможную угрозу. А вы не должны просто снимать помеченность с любых получаемых данных и отправлять их на встроенную `eval`. (Если вы это сделаете, то *мы* почувствуем соблазн взломать вашу систему и удалить этот сценарий.)

Здесь приходит на помощь `reval`. Вот сценарий CGI, который обрабатывает форму с единственным полем, вычисляет (в скалярном контексте) строку, которую в нем найдет, и выводит отформатированный результат:

```
#!/usr/bin/perl -lTw
use strict;
use CGI::Carp 'fatalsToBrowser';
use CGI qw/:standard escapeHTML/;
use Safe;

print header(-type => "text/html;charset=UTF-8"),
      start_html("Perl Expression Results");
```

[1] Не надо смеяться. Мы действительно встречали веб-страницы, которые это делают. Без `Safe`!

```
my $expr = param("EXPR") =~ /^([^;]+)/
            ? $1 # возвращает часть, теперь уже не меченую
            : croak("no valid EXPR field in form");
my $answer = Safe->new->reval($expr);
die if $@;

print p("Result of", tt(escapeHTML($expr)),
                "is", tt(escapeHTML($answer)));
```

Теперь представьте себе зловредного пользователя, который введет строку `"print `cat /etc/passwd`"` (или еще худшую). Благодаря окружению с наличием ограничений, которое запрещает обратные слэши, Perl распознает проблему во время компиляции и немедленно возвращается. В переменной `$@` будет строка `"quoted execution (``, qx) trapped by operation mask"` (исполнение в кавычках, перехваченное маской операций) плюс завершающая информация о том, где возникла эта проблема.

Поскольку мы не указали иного, создававшиеся нами защищенные разделы использовали установленный по умолчанию набор разрешенных операций. Сейчас не важно, каким образом задаются разрешенные или запрещенные операции. Важно, что это полностью находится под контролем вашей программы. А поскольку в программе можно создать несколько объектов `Safe`, то допустимо назначать разные степени доверия разным фрагментам кода в зависимости от источника их получения.

Если вы хотите поэкспериментировать с `Safe`, то вот маленький интерактивный калькулятор на Perl. Это калькулятор, потому что можно вводить в него числовые выражения и немедленно получать их значения. Но он не ограничен только числами. Он большее похож на пример цикла в описании `eval` из главы 29 «Функции» – там можно было взять все что угодно, вычислить это и получить результат. Разница в том, что версия с `Safe` не выполняет все что угодно. Вы можете запустить этот калькулятор интерактивно на своем терминале, вводя фрагменты кода Perl и просматривая результаты, чтобы почувствовать, какого рода защиту предоставляет `Safe`.

```
#!/usr/bin/perl -w
# safecalc - демо-программа для проверки Safe
use strict;
use Safe;
my $sandbox = Safe->new();
while (1) {
    print "Input: ";
    my $expr = <STDIN>;
    exit unless defined $expr;
    chomp($expr);
    print "$expr produces ";
    local $SIG{__WARN__} = sub { die @_ };
    my $result = $sandbox->reval($expr, 1);
    if ($@ =~ s/at \(eval \d+\).*//) {
        printf "[%s]: %s", $@ =~ /trapped by operation mask/
            ? "Security Violation" : "Exception", $@;
```

```
        }
        else {
            print "[Normal Result] $result\n";
        }
    }
```

Предупреждение: модуль Safe в настоящее время перерабатывается, с тем чтобы каждый раздел выполнялся в совершенно независимом интерпретаторе Perl в рамках того же самого процесса. (Такой принцип использует mod_perl для Apache при запуске компилированных сценариев Perl.) В настоящее время детали еще не ясны, но наш хрустальный шар показывает, что после того как модуль будет переписан, вы недалеко уйдете, пытаясь заглянуть в раздел с помощью именованного пакета. Если у вас Perl версии новее, чем 5.6, посмотрите в примечаниях к выпуску в *perldelta*(1), что изменилось, или ознакомьтесь с документацией по самому модулю Safe. (Конечно, вы и так всегда это делаете, не правда ли?)

Код, маскирующийся под данные

Разделами Safe можно пользоваться, когда загружаются действительно опасные данные, но это не значит, что к защите можно совсем не прибегать в повседневной работе по дому. Необходимо совершенствовать свое знание окружающей местности и смотреть на вещи с точки зрения субъекта, желающего проникнуть внутрь. Необходимо предпринимать упреждающие меры, например, поддерживать хорошее освещение и подстригать кусты, в которых могут притаиться проблемы.

Perl тоже старается помочь вам в этом деле. Принятая в Perl схема синтаксического анализа и выполнения позволяет избежать ошибок, жертвами которых часто становятся языки программирования оболочек. Язык обладает многими очень мощными возможностями, но они умышленно синтаксически и семантически ограничены таким образом, чтобы все находилось под контролем программиста. За немногими исключениями, Perl вычисляет каждую лексему только один раз. Если похоже, что нечто используется как простая переменная данных, оно не начнет вдруг шарить в вашей файловой системе.

К сожалению, такого рода вещи могут случиться, если вы обратитесь к оболочке для запуска других программ, потому что тогда вы будете работать по правилам оболочки, а не Perl. Хотя применения оболочки легко избежать – нужно лишь использовать формат функций system, exec или open с открытием канала, в котором передается список аргументов. Хотя для обратных слэшей нет формата со списком аргументов, защищающего от оболочки, всегда можно эмулировать их, как описано в разделе «Доступ к командам и файлам при ограниченных правах». (Нет синтаксического способа заставить обратные слэши принимать список аргументов, но сейчас разрабатывается формат с многими аргументами, лежащий в основе оператора readpipe; на момент написания книги он еще не вполне был готов.)

Если вы используете переменную в выражении (в том числе интерполируете ее в строку в двойных кавычках), исключается возможность того, что эта переменная содержит код Perl, который делает нечто, вами не предусмотренное.[1] В отличие от оболочки, Perl не требует охранительных кавычек вокруг переменных, что бы в них ни содержалось.

```
$new = $old;                # Кавычки не требуются.
print "$new items\n";       # $new не навредит вам.

$phrase = "$new items\n";   # И здесь тоже.
print $phrase;              # По-прежнему ok.
```

В Perl реализован подход «what you see is what you get» (что видишь, то и имеешь). Если вы не видите дополнительного уровня интерполяции, значит, его нет. *Можно* интерполировать в строки произвольные выражения Perl, но только если специально попросить об этом Perl. (И даже в этом случае содержимое подвергается проверке на чистоту данных, если включен этот режим.)

```
$phrase = "You lost @{[ 1 + int rand(6) ]} hit points\n";
```

Однако интерполяция не является рекурсивной. Нельзя спрятать в строке произвольное выражение:

```
$count = '1 + int rand(6)';             # Некоторый случайный код.
$saying = "$count hit points";          # Просто литерал.
$saying = "@{[$count]} hit points";     # Тоже литерал.
```

Оба присваивания `$saying` создают `"1 + int rand(6) hit points"`, не вычисляя интерполированное содержание `$count` как код. Чтобы заставить Perl это сделать, нужно явно вызвать `eval` *`STRING`*:

```
$code = '1 + int rand(6)';
$die_roll = eval $code;
die if $@;
```

Если бы `$code` была меченой, то `eval` *`STRING`* создала бы исключительную ситуацию. Конечно, почти никогда не требуется вычислять случайный код, но если вдруг придется, то следует подумать об использовании модуля `Safe`. Возможно, вы уже слышали о таком.

Есть одно место, в котором Perl может иногда обрабатывать данные как код, а именно, когда шаблон в операторах `qr//`, `m//` или `s///` содержит одно из новых утверждений регулярных выражений, `(?{`*`CODE`*`})` или `(??{`*`CODE`*`})`. Они не угрожают безопасности, если используются как литералы при поиске по шаблону:

---

[1] Хотя, если вы генерируете веб-страницу, можно получить на выходе теги HTML, в том числе код JavaScript, которые сделают нечто, чего удаленный броузер не ожидает.

```
$cnt = $n = 0;
while ($data =~ /( \d+ (?{ $n++ }) | \w+ )/gx) {
    $cnt++;
}
print "Got $cnt words, $n of which were digits.\n";
```

Но существующий код, который интерполирует переменные в поиск, был написан в предположении, что данные – это данные, а не код. Эти новые конструкции могли создать брешь в защите программ, которые прежде были безопасными. Поэтому Perl отказывается вычислять шаблон, если интерполируемая строка содержит утверждение кода и создает исключительную ситуацию. Если такая функция вам действительно требуется, всегда можно включить ее с помощью директивы `use re 'eval'` с лексической областью видимости. (Однако по-прежнему нельзя использовать меченые данные для интерполируемого утверждения кода.)

Совершенно иного рода проблемы безопасности, которые могут быть связаны с регулярными выражениями, относятся к отказу в обслуживании. Из-за них программа может закончить работу преждевременно или работать слишком долго, или исчерпать всю имеющуюся память, а иногда и вывести дамп памяти – в зависимости от фазы луны.

При обработке шаблонов, вводимых пользователями, не нужно беспокоиться о возможности интерпретации случайного кода Perl. Однако в механизме регулярных выражений есть свои маленькие компилятор и интерпретатор, а шаблон, введенный пользователем, может вызвать у компилятора регулярных выражений изжогу. Если интерполированный шаблон не является допустимым шаблоном, создается исключительная ситуация, которая окажется фатальной, если не будет перехвачена. Организуя ее перехват, используйте только `eval` *`BLOCK`*, а не `eval` *`STRING`*, потому что дополнительный уровень вычисления последнего может на практике допустить выполнение случайного кода Perl. Вместо этого сделайте нечто вроде следующего:

```
if (not eval { "" =~ /$match/; 1 }) {
    # (Здесь подвергните подозрительный шаблон необходимой обработке.)
}
else {
    # Мы знаем, что шаблон, по крайней мере, компилируется.
    if ($data =~ /$match/) { ... }
}
```

Более неприятная проблема отказа в обслуживании возникает, когда при правильных данных и правильном шаблоне поиска программа оказывается зависшей навсегда. Это связано с тем, что для некоторых шаблонов поиска время работы растет экспоненциально и может превысить среднее время наработки на отказ нашей солнечной системы. Если вам особенно повезет, то для такого шаблона с интенсивными вычислениями потребуется еще и экспоненциально растущий объем памяти. В таком случае ваша программа исчерпает всю имеющуюся виртуальную память, утопит остальную часть системы, выведет из себя пользователей и либо закончит существование

с обычной ошибкой «`Out of memory!`», либо оставит после себя громадный файл с дампом памяти, хотя, возможно, он будет не таким большим, как солнечная система.

Как и с большинством атак типа «отказ в обслуживании», с этой проблемой нелегко справиться. Если платформа поддерживает функцию `alarm`, можно установить тайм-аут для поиска по шаблону. К сожалению, Perl не может (в настоящее время) гарантировать, что простой акт обработки сигнала не вызовет дампа памяти. (Планируется исправить это в следующих версиях.) Однако вы всегда можете попробовать этим воспользоваться, и даже если сигнал не удастся красиво обработать, по крайней мере, программа не будет работать вечно.

Если в вашей системе поддерживается ограничение ресурсов, выделенных каждому процессу, можете установить их из оболочки, прежде чем запускать программу Perl, либо использовать модуль `BSD::Resource` из CPAN, чтобы сделать это прямо из Perl. Веб-сервер Apache позволяет устанавливать границы времени, памяти и размера файла для запускаемых им сценариев CGI.

Наконец, мы надеемся, что у вас осталось некоторое неразвеявшееся чувство небезопасности. Помните, что если вы параноик, то это не значит, что за вами никто не гонится. Можете даже получить от этого удовольствие.

24

Распространенные приемы программирования

Спросите любого программиста Perl, и он будет рад дать вам массу советов о том, как программировать. Мы не составляем исключения (как вы могли заметить). В данной главе мы не станем сообщать вам о специфических возможностях Perl, а зайдем с другого конца и используем более вольный подход для описания идиом Perl. Надеемся, что, сведя вместе различные вещи, которые внешне кажутся несвязанными, вы сможете в большей мере проникнуться ощущением того, что означает «думать на Perl». В конце концов, когда вы программируете, вы не пишете несколько выражений, затем несколько подпрограмм, затем несколько объектов. В какой-то мере вы должны заниматься всем этим одновременно. Вот и в этой главе – немного обо всем.

Однако элементарная организация здесь все-таки есть, и проявится она в том, что, начав с отрицательных примеров, мы будем продвигаться в направлении положительных. Не знаем, станет ли от этого легче вам, но нам – станет.

Обычные промахи новичков

Самая большая ошибка, которую совершают все, – это отключение вывода предупреждений по директиве `use warnings`, идентифицирующей многие ошибки. Вторая по важности ошибка – забыть `use strict` в нужных местах. Эти две прагмы могут уберечь вас от долгих часов бесплодных раздумий, когда ваша программа начнет разрастаться в объеме. (А она начнет.) Еще

одним ложным шагом будет забыть проконсультироваться с электронным FAQ. Допустим, вы хотите узнать, есть ли в Perl функция round. Сначала можно попытаться поискать в FAQ:

```
% perlfaq round
```

Помимо этих «метапромахов» есть несколько типов ловушек программирования. В некоторые из них попадает почти каждый, а в другие можно угодить, только придя из другой культуры, где многое делается иначе. Ошибки эти сгруппированы в нижеследующих разделах.

Всеобщие ошибки

- Ввод запятой после указателя файла в команде print. Хотя кажется совершенно нормальным и приличным сказать:

  ```
  print STDOUT, "goodbye", $adj, "world!\n";    # НЕВЕРНО
  ```

  это, тем не менее, некорректно из-за этой первой запятой. Вместо этого здесь нужен синтаксис косвенного объекта:

  ```
  print STDOUT "goodbye", $adj, "world!\n";     # ok
  ```

  Синтаксис этот позволяет сказать:

  ```
  print $filehandle "goodbye", $adj, "world!\n";
  ```

  где $filehandle представляет собой скаляр, содержащий указатель файла на этапе исполнения. Это не то же самое, что:

  ```
  print $notafilehandle, "goodbye", $adj, "world!\n";
  ```

  где $notafilehandle является просто строкой, представляющей часть списка выводимых вещей. См. «косвенный объект» в глоссарии.

- Применение == вместо eq и != вместо ne. Операторы == и != используются для проверки *чисел*. Другие два оператора выполняют проверки *строк*. Строки "123" и "123.00" равны как числа, но не равны как строки. Кроме того, любая нечисловая строка численно равна нулю. Если только вы не работаете с числами, вам почти всегда требуются операторы сравнения строк.

- Забывают ввод замыкающей точки с запятой. Каждая команда в Perl завершается точкой с запятой или концом блока. Символы перевода строки не служат окончанием команды, как в *awk*, Python или FORTRAN. Помните, что Perl похож на C.

  Команда, содержащая встроенный документ, особенно подвержена потере точки с запятой. Она должна выглядеть так:

  ```
  print <<'FINIS';
  A foolish consistency is the hobgoblin of little minds,
  adored by little statesmen and philosophers and divines.
                                        - -Ralph Waldo EmersonFINIS
  ```

- Забывают, что *BLOCK* требует фигурных скобок. Простые команды не являются блоками. При создании управляющей структуры, такой как `while` или `if`, состоящей из одного или более блоков, *необходимо* использовать фигурные скобки вокруг каждого блока. Помните, что Perl – это *не* C.
- Забывают сохранять переменные `$1`, `$2` и т. д. при переходе к новому РВ. Помните, что каждый очередной `m/atch/` или `s/ubsti/tution/` изменяет (или чистит, или калечит) переменные `$1`, `$2`..., равно как `$`, `$&`, `$'` со товарищи. Один из способов сразу сохранить их – выполнить поиск в списковом контексте:

```
my ($one, $two) = /(\w+) (\w+)/;
```

- Многие не понимают, что `local` изменяет значение переменной, которое видят другие подпрограммы, вызываемые в пределах области видимости локальной переменной. Легко забыть, что `local` является командой этапа исполнения, которая создает динамическую область видимости, поскольку для нее нет эквивалента в языках типа C. См. раздел «Объявления с областью видимости» в главе 4 «Команды и объявления». В любом случае обычно требуется `my`.
- Потеря парности фигурных скобок. Хорошие текстовые редакторы помогают находить пары. Обзаведитесь таким. (Или такими.)
- Применение команд управления циклами в `do {} while`. Хотя фигурные скобки в этой управляющей структуре выглядят подозрительно похоже на часть блока цикла, они таковой не являются.
- Использование `$foo[1]`, когда имеется в виду `$foo[0]`. Perl по умолчанию всегда начинает нумерацию с нуля.
- Запись `@foo[0]`, когда имеется в виду `$foo[0]`. Ссылка `@foo[0]` является срезом массива, т. е. массивом, состоящим из единственного элемента `$foo[0]`. Иногда никакой разницы нет, как в:

```
print "the answer is @foo[0]\n";
```

  но она очень велика в следующем случае:

```
@foo[0] = <STDIN>;
```

  когда будет «проглочено» все, оставшееся в `STDIN`, переменной `$foo[1]` будет присвоена *первая* строка, а все остальное будет отброшено. Едва ли это входило в ваши намерения. Привыкните думать, что `$` соответствует единственному значению, а `@` – списку значений, и все у вас будет в порядке.
- Потеря круглых скобок в списковом операторе типа `my`:

```
my  $x, $y  = (4, 8);     # НЕВЕРНО
my ($x, $y) = (4, 8);     # ok
```

- Иногда забывают выбрать правильный указатель файла, прежде чем установить `$^`, `$~` или `$|`. Эти переменные зависят от указателя файла,

выбранного в данный момент согласно select(*FILEHANDLE*). Первоначальным указателем файла, выбранным в данный момент, является STDOUT. Вместо этого следует использовать методы указателя файла из модуля FileHandle. См. главу 28 «Специальные имена».

Часто игнорируемые советы

Практикующие программисты Perl должны учесть следующее:

- Помните, что многие операции по-разному ведут себя в списковом и скалярном контекстах. Например:

```
($x) = (4, 5, 6);        # Списковый контекст; $x устанавливается в 4
$x  = (4, 5, 6);         # Скалярный контекст; $x устанавливается в  6

@a  = (4, 5, 6);
$x  = @a;                # Скалярный контекст; $x устанавливается в 3
                         #  (длина массива)
```

- По возможности избегайте голых (bare) слов, особенно если они написаны целиком в нижнем регистре. Нельзя, лишь посмотрев на слово, сказать, функция это или просто строка. Используя кавычки для строк и круглые скобки для аргументов при вызове функции, вы никогда их не спутаете. На практике директива use strict в начале программы делает голые слова ошибкой этапа компиляции, что, вероятно, правильно.
- Невозможно по внешнему виду сказать, какие встроенные функции являются унарными операторами (как chop и chdir), какие – списковыми (как print и unlink), а какие не имеют аргументов (как time). Это вы сможете узнать из главы 29 «Функции». Как всегда, если вы не уверены, используйте круглые скобки – даже если вы не уверены в том, что вы не уверены. Заметьте также, что определенные пользователем подпрограммы по умолчанию представляют собой списковые операторы, но они могут быть объявлены как унарные операторы с помощью прототипа ($)или как не имеющие аргументов с помощью прототипа ().
- Трудно запомнить, что некоторые функции по умолчанию имеют в качестве аргументов $_, @ARGV или что-либо еще, а другие – нет. Потратьте время, чтобы запомнить «кто есть кто», либо избегайте использования аргументов по умолчанию.
- <FH> является не указателем файла; это оператор угловых скобок, осуществляющий операцию ввода строк в указатель. Эта путаница обычно проявляется, когда пытаются выполнять print в этот оператор:

```
print <FH> "hi";    # НЕВЕРНО, опустите угловые скобки
```

- Помните также, что данные, которые читает оператор угловых скобок, присваиваются $_, только когда чтение файла является единственным условием в цикле while:

```
while (<FH>) { }   # Данные присваиваются $_.
```

```
<FH>;                # Данные читаются и отбрасываются!
```

- Не применяйте =, когда требуется =~; эти две конструкции совершенно различны:

```
$x =  /foo/;  # Ищет в $_ "foo", помещает результат в $x
$x =~ /foo/;  # Ищет в $x "foo", отбрасывает результат
```

- Используйте my для локальных переменных, если можно обойтись этим. Ключевое слово local просто дает временное значение глобальной переменной, что оставляет вас открытым воздействию непредвиденных побочных эффектов динамических областей видимости.
- Избегайте применения local с экспортированными переменными модуля. Если вы локализуете экспортированную переменную, ее экспортированное значение не изменится. Локальное имя становится псевдонимом для нового значения, но внешнее имя остается псевдонимом оригинала.

Ловушки C

Высоколобые программисты на C должны учесть следующее:

- В блоках if и while требуются фигурные скобки.
- Вы должны использовать elsif, а не «else if» или «elif». Следующий синтаксис:

```
if (expression) {
 block;
}
else if (another_expression) {      # НЕВЕРНО
    another_block;
}
```

  является недопустимым. Часть else всегда является блоком, а голый if не является блоком. Не ждите от Perl точного сходства с C. Вот что вам нужно:

```
if (expression) {
    block;
}
elsif (another_expression) {
    another_block;
}
```

  Обратите также внимание, что «elif» – это перевернутый «file». Только пользователям Algol могло понадобиться ключевое слово, которое совпадает с другим словом, записанным наоборот.

- Ключевые слова break и continue из C становятся в Perl last и next, соответственно. В отличие от C, они *не* работают в конструкции do {} while.
- Отсутствует команда switch. (Но ее легко построить на лету; см. «Голые блоки» и «Структуры типа Case» в главе 4.)

- Переменные начинаются в Perl с $, @ или %.
- Комментарии начинаются с #, а не /*.
- Нельзя получить адреса чего-либо, хотя в Perl есть аналогичный оператор (обратный слэш), создающий ссылку.
- ARGV должен записываться на верхнем регистре. $ARGV[0] — это argv[1] в C, а argv[0] из C приводит к $0.
- Системные вызовы, такие как link, unlink и rename, возвращают в случае успеха истину, а не 0.
- Обработчики сигналов в %SIG работают с именами, а не номерами сигналов.

Ловушки оболочки

Сообразительные программисты оболочек (shells) должны принять во внимание следующее:

- Переменные в левой части присваивания имеют префиксы $, @ или %, как и в правой. Присваивание в стиле оболочки:

  ```
  camel='dromedary';      # НЕВЕРНО
  ```

  будет проанализировано не так, как вам бы хотелось. Правильно так:

  ```
  $camel='dromedary';     # ok
  ```

- Переменная цикла в foreach тоже требует $. Тогда как в *csh* допустимо:

  ```
  foreach hump (one two)
      stuff_it $hump
  end
  ```

  в Perl это записывается так:

  ```
  foreach $hump ("one", "two") {
      stuff_it($hump);
  }
  ```

- Оператор обратного штриха производит интерполяцию, не обращая внимания на присутствие одинарных кавычек в команде.
- Оператор обратного штриха не производит трансляцию возвращаемого значения. В Perl необходимо отбросить символ перевода строки явно, например так:

  ```
  chomp($thishost = `hostname`);
  ```

- Оболочки (особенно *csh*) выполняют несколько уровней подстановки в каждой командной строке. Perl осуществляет интерполяцию только в определенных конструкциях, таких как двойные кавычки, обратные штрихи, угловые скобки и шаблоны для поиска.

- Оболочки имеют тенденцию выполнять сценарии по частям. Perl целиком компилирует программу перед ее выполнением (за исключением блоков BEGIN, которые выполняются до того, как завершена компиляция).
- Доступ к аргументам программы осуществляется через @ARGV, а не $1, $2 и т. д.
- Доступ к окружению как к отдельным скалярным переменным не предоставляется автоматически. Если вам это требуется, используйте стандартный модуль Env.

Ловушки прежнего Perl

Кающиеся программисты Perl 4 (и предыдущих версий) должны обратить внимание на следующие изменения в версии 5 по сравнению с версией 4, которые могут повлиять на старые сценарии:

- @ теперь всегда интерполирует массив в строках, заключенных в двойные кавычки. Некоторым программам теперь потребуется использовать обратный слэш для защиты тех @, которые не должны интерполироваться.
- Простые слова, которые выглядели для Perl как строки, теперь выглядят как вызовы подпрограмм, если компилятору ранее встречались определения подпрограмм с такими же именами. Например:

  ```
  sub SeeYa { die "Hasta la vista, baby!" }
  $SIG{'QUIT'} = SeeYa;
  ```

  В прежних версиях Perl этот код устанавливал бы обработчик сигнала. Теперь он вызывает эту функцию! Можно использовать ключ -w, чтобы обнаружить такое рискованное употребление, или директиву use strict, чтобы запретить его.
- Идентификаторы, начинающиеся с символа подчеркивания «_», больше не помещаются в пакет main, за исключением переменных, состоящих из одного подчеркивания ($_, @_ и т. д.).
- Удвоенное двоеточие является теперь действующим разделителем пакета в идентификаторе. Например, в команде:

  ```
  print "$a::$b::$c\n";
  ```

  теперь $a:: рассматривается как ссылка на переменную, тогда как в прежних версиях только $a рассматривалась как ссылка на переменную. Аналогично:

  ```
  print "$var::abc::xyz\n";
  ```

  теперь интерпретируется как одна переменная $var::abc::xyz, тогда как в прежних версиях за переменной $var следовала бы текстовая константа ::abc::xyz.

- `s'$pattern'replacement'` теперь не осуществляет интерполяции в `$pattern`. (`$` следует интерпретировать как утверждение конца строки.) Такой режим действует только при использовании одинарных кавычек в качестве разделителя в подстановке; в других подстановках `$pattern` всегда интерполируется.
- Второй и третий аргументы `splice` теперь вычисляются в скалярном контексте, а не в списковом.
- Теперь следующий код считается семантически ошибочным из-за приоритетов:

```
shift @list + 20;    # Теперь анализируется как shift(@list + 20), недопустимо!
$n = keys %map + 20; # Теперь анализируется как keys(%map + 20), недопустимо!
```

  Поскольку если бы эти команды работали, не работала бы следующая:

```
sleep $dormancy + 20;
```

- Приоритет операторов присваивания теперь такой же, как у присваивания. В прежних версиях Perl им ошибочно давался приоритет связанного с ними оператора. Поэтому теперь вы должны заключать их в скобки в выражениях типа:

```
/foo/ ? ($a += 2) : ($a -= 2);
```

  В противном случае:

```
/foo/ ? $a += 2 : $a -= 2;
```

  было бы ошибочно проанализировано как:

```
(/foo/ ? $a += 2 : $a) -= 2;
```

  С другой стороны,

```
$a += /foo/ ? 1 : 2;
```

  теперь работает так, как ожидал бы программист C.
- Команда `open FOO || die` неверна. Необходимо заключить указатель файла в скобки, потому что `open` имеет приоритет спискового оператора.
- Элементы списков аргументов для форматов теперь вычисляются в списковом контексте. Это означает, что теперь можно интерполировать списковые значения.
- Нельзя использовать `goto` для перехода в блок, удаленный при оптимизации. Черт возьми.
- Больше не допускается использовать пробельный символ в качестве имени переменной или как разделитель в конструкции кавычек. Дважды черт возьми.

- Функция `caller` теперь возвращает ложное значение в скалярном контексте, если вызывающего нет. Это позволяет модулям определить, загружены они через `require` или запущены непосредственно.
- `m//g` теперь связывает свое состояние со строкой, в которой производится поиск, а не с регулярным выражением. Подробности можно найти в главе 5 «Поиск по шаблону».
- `reverse` теперь не допускается использовать в качестве имени подпрграммы в `sort`.
- *taintperl* больше не является отдельной программой. Теперь есть ключ `-T` для активации режима меченых данных, если он не включается автоматически.
- Строки в двойных кавычках теперь не могут оканчиваться `$` или `@` без `\` перед ними.
- Архаический синтаксис `if` *`BLOCK BLOCK`* более не поддерживается.
- Отрицательные индексы массивов теперь отсчитываются от конца массива.
- Оператор запятой в скалярном контексте теперь обеспечивает скалярный контекст для своих аргументов.
- Оператор `**` теперь связывает более сильно, чем унарный минус.
- Уменьшение `$#array` теперь немедленно выбрасывает элементы массива.
- `delete` не обязательно возвращает удаленное значение для связанных массивов, поскольку некоторым модулям затруднительно реализовать такую возможность.
- Конструкция `"this is $$x"`, которая интерполировала в данной точке ID процесса, теперь пытается разыменовать `$x`. Однако сам по себе `$$` по-прежнему работает нормально.
- Поведение `foreach` при обходе списка, не являющегося массивом, слегка изменилось. Раньше список присваивался временному массиву, но теперь для повышения эффективности не делает этого. Это означает, что теперь выполняется обход фактических значений, а не их копий. Модификация переменной цикла может изменить исходные значения, даже после `grep`! Например:

```
% perl4 -e '@a = (1,2,3); for (grep(/./, @a)) { $_++ }; print "@a\n"'
1 2 3
% perl5 -e '@a = (1,2,3); for (grep(/./, @a)) { $_++ }; print "@a\n"'
2 3 4
```

- Чтобы сохранить прежнюю семантику Perl, потребуется явно присвоить свой список временному массиву и выполнять его обход. Например, может понадобиться изменить

```
foreach $var (grep /x/, @list) { ... }
```

на:

```
foreach $var (my @tmp = grep /x/, @list) { ... }
```

В противном случае изменение $var разрушит значения в @list. (Чаще всего это происходит при использовании $_ в качестве переменной цикла и вызове в цикле подпрограмм, которые не локализуют должным образом $_.)

- Некоторые сообщения об ошибках или предупреждения могут быть иными.
- Некоторые ошибки могут оказаться непреднамеренно ликвидированными.

Эффективность

Хотя большая часть работы программирования может состоять лишь в том, чтобы заставить программу правильно работать, не исключено, что вы захотите добиться большего от своей программы на Perl. Богатый арсенал операторов, типов данных и управляющих конструкций в Perl не всегда позволяет интуитивно оценить возможности оптимизации по скорости или памяти. Разработчикам Perl пришлось пойти на многочисленные компромиссы, и такие решения погребены в глубине кода. В целом, чем короче и проще ваш код, тем он быстрее выполняется, но есть и исключения. В данном разделе предпринята попытка помочь вам заставить его работать капельку лучше.

Если вы хотите заставить его работать намного лучше, поэкспериментируйте с сервером компилятора Perl, описанным в главе 18 «Компиляция», или переписать внутренний цикл как расширение C, что продемонстрировано в главе 21 «Внутри и снаружи».

Обратите внимание, что оптимизация по времени может иногда обернуться нерациональным расходованием памяти или снижением эффективности программирования (на что указывают противоречивые советы ниже). Если бы программирование было простым делом, для него не потребовалось бы такое сложное создание, как человек, не так ли?

Эффективность по времени

- Используйте хеши вместо линейного поиска. Например, вместо поиска по @keywords, чтобы узнать, является ли $_ ключевым словом, создайте хеш следующим образом:

```
my %keywords;
for (@keywords) {
    $keywords{$_}++;
}
```

После этого можно быстро сказать, содержит ли $_ ключевое слово, проверив $keyword{$_} на отличие от нуля.

- Избегайте употребления индексов, когда можно использовать оператор foreach или список. Применение индексов не только требует дополнительных операций, но и, если ваши индексы (поскольку вы их вычисляли) окажутся числами с плавающей точкой, то потребуется дополнительное преобразование из плавающей точки обратно в целые числа. Часто можно отыскать лучший способ сделать это. Обдумайте возможность применения foreach, shift и splice. Попробуйте сказать use integer.
- Избегайте использования оператора goto. Он производит сканирование кода от вашего текущего адреса в поисках заданной метки.
- Избегайте printf, если можно обойтись print.
- Избегайте $& и дружественных $` и $'. Их появление в программе приводит к тому, что при любом успешном поиске строка, в которой производится поиск, сохраняется для возможного использования в будущем. (Однако, если они появились один раз, хуже не будет, если их будет больше.)
- Избегайте применения eval к строке. eval строки (а не *BLOCK*) требует при каждом вызове перекомпиляции. Анализатор Perl работает довольно быстро, но это ни о чем не говорит. Сейчас почти всегда есть лучший способ сделать то, что вам требуется. В частности, любой код, в котором eval служит только для создания имен переменных, является устаревшим, поскольку то же самое можно сделать непосредственно с помощью символических ссылок:

```
no strict 'refs';
$name = "variable";
$$name = 7;           # устанавливает $variable равной 7
```

- Избегайте eval *STRING* в цикле. Вместо этого поместите цикл в eval, чтобы избежать избыточной перекомпиляции кода. В качестве примера посмотрите на оператор study в главе 29.
- Избегайте шаблонов, компилируемых на этапе исполнения. Примените модификатор шаблона */pattern/o* (однократно), чтобы избежать перекомпиляции шаблона, который не меняется в течение жизни процесса. С шаблонами, которые изредка изменяются, можно использовать то обстоятельство, что нулевой шаблон ссылается на предшествующий шаблон, например:

```
"foundstring" =~ /$currentpattern/;   # Фиктивный поиск (должен быть успешным).
while (<>) {
    print if //;
}
```

Другой возможностью является предварительная компиляция регулярного выражения с помощью конструкции кавычек в qr. Допускается также использовать eval для повторной компиляции подпрограммы, осуществляющей поиск (если рекомпиляция происходит редко). Еще лучше скомпилировать несколько поисков в одну подпрограмму, амортизируя тем самым расходы на вызов подпрограмм.

- Оператор короткого действия для альтернатив часто работает быстрее, чем соответствующее регулярное выражение, поэтому

  ```
  print if /one-hump/ || /two/;
  ```

  будет, вероятно, быстрее, чем

  ```
  print if /one-hump|two/;
  ```

  по крайней мере, для некоторых значений `one-hump` и `two`. Последнее связано с тем, что оптимизатор любит поднимать некоторые простые операции поиска в верхние части синтаксического дерева и осуществлять очень быстрый поиск по алгоритму Бойера-Мура (Boyer-Moore). Сложный шаблон может помешать этому.

- Предварительно исключайте из рассмотрения «обычные» последовательности с помощью `next if`. Это, как и простые регулярные выражения, оптимизатор любит. И это разумно, чтобы избежать лишней работы. Обычно можно выкинуть строки комментариев и пустые строки, прежде чем выполнять `split` или `chop`:

  ```
  while (<>) {
      next if /^#/;
      next if /^$/;
      chop;
      @piggies = split(/,/);
      ...
  }
  ```

- Избегайте регулярных выражений с большим количеством квантификаторов и большими числами {*MIN*, *MAX*} для выражений в скобках. Такие шаблоны могут привести к экспоненциально медленному поиску с возвращением, если их «квантифицированные подшаблоны» не будут найдены при первом «проходе». Можно также использовать конструкцию `(?>...)`, чтобы потребовать полного соответствия шаблона либо признания неудачи без осуществления поиска с возвращением.

- Старайтесь максимально увеличить длину всех не необязательных литеральных строк в регулярных выражениях. Интуиция подсказывает обратное, но длинные шаблоны часто отыскиваются скорее, чем короткие, – из-за того, что оптимизатор ищет постоянные строки и передает их алгоритму Бойера-Мура, который выигрывает от длинных строк. Скомпилируйте шаблон с ключом отладки `-Dr` и посмотрите, какую строку литералов Dr. Perl считает самой длинной.

- Избегайте дорогостоящего вызова подпрограмм в интенсивных циклах. С вызовом подпрограмм связаны накладные расходы, особенно при передаче длинных списков параметров и возврате длинных значений. В порядке возрастания степени отчаяния попробуйте передавать значения по ссылке, передавать значения как глобальные переменные с динамической областью видимости, встраивать (inlining) подпрограмму или переписать

весь цикл на C. (Лучше всего, если вы сможете вообще избавиться от подпрограмм, применив более умный алгоритм.)

- Избегайте обращения к функции `getc` кроме как для ввода одиночных символов с терминала. На самом деле лучше и для этого ее не использовать. Следует предпочесть `sysread`.
- Избегайте частого применения `substr` с длинными строками, особенно если строка содержит UTF-8. Эту функцию лучше использовать в начале строки, а в некоторых задачах можно сделать так, чтобы `substr` работала с началом, «зажевывая» строку по ходу дела с помощью `substr` с четырьмя аргументами путем замены захваченной части на `""`:

```
while ($buffer) {
    process(substr($buffer, 0, 10, ""));
}
```

- Используйте `pack` и `unpack` вместо многократного вызова `substr`.
- Применяйте `substr` в качестве левого значения вместо конкатенации подстрок. Например, для замены символов `$foo` с четвертого по седьмой содержимым переменной `$bar` не делайте так:

```
$foo = substr($foo,0,3) . $bar . substr($foo,7);
```

Вместо этого просто укажите заменяемую часть строки и выполните присваивание ей:

```
substr($foo, 3, 4) = $bar;
```

Но помните, что если `$foo` является очень большой строкой, а `$bar` не соответствует в точности размеру «дырки», то это может привести к большому объему копирования. Perl попытается минимизировать его, выбрав копирование с начала или с конца, но больше ничего не сможет сделать, если `substr` находится в середине.

- Используйте `s///` вместо конкатенации подстрок. Особенно это относится к случаю, когда можно заменить одну константу другой такого же размера. В результате получится подстановка по месту.
- Применяйте модификаторы команд и эквивалентные операторы `and` и `or` вместо полноразмерных условных операторов. Модификаторы команд (типа `$ring = 0 unless $engaged`) и логические операторы позволяют избежать расходов, связанных с входом и выходом из блока. Часто они также лучше читаются.
- Используйте `$foo = $a || $b || $c`. Это значительно быстрее (и короче), чем:

```
if ($a) {
    $foo = $a;
}
elsif ($b) {
```

```
        $foo = $b;
    }
    elsif ($c) {
        $foo = $c;
    }
```

Аналогично, устанавливайте значения по умолчанию с помощью

```
$pi ||= 3;
```

- Группируйте все проверки, требующие одинаковой начальной строки. Проверяя строки на различные префиксы при помощи какой-либо конструкции, напоминающей структуру переключателя, соберите вместе все шаблоны /^a/, все шаблоны /^b/ и т. д.
- Избегайте проверок, если известно, что они закончатся неудачей. Используйте last или elsif, чтобы избежать «провала» в следующую секцию вашего переключателя.
- Используйте специальные операторы, такие как study, логические операции над строками, форматы pack 'u' и unpack '%'.
- Помните о хвосте, который машет собакой. Ошибочные команды – что-то вроде (<STDIN>)[0] – могут перегрузить Perl лишней работой. В соответствии с философией Unix, Perl предоставляет пользователю веревку, достаточно длинную для того, чтобы на ней можно было повеситься.
- Выносите операции за циклы. Оптимизатор Perl не пытается переместить инвариантный код из циклов. Он рассчитывает, что вы проявите некоторый разум.
- Строки могут быть быстрее массивов.
- Массивы могут быть быстрее строк. Все зависит от того, предполагается ли повторно использовать строки или массивы, и какие операции предполагается выполнять. Значительная модификация каждого элемента означает, что лучше использовать массивы, а эпизодическая модификация некоторых элементов означает, что следует предпочесть строки. Но вы должны попробовать и убедиться в этом сами.
- Переменные my быстрее, чем переменные local.
- Сортировка по созданному вручную массиву ключей может оказаться быстрее, чем использование замысловатой подпрограммы сортировки. Значение массива обычно сравнивается несколько раз, поэтому если подпрограмма сортировки должна производить много пересчета, лучше вывести это вычисление в отдельный проход перед фактической сортировкой.
- Если выполняется удаление символов, то tr/abc//d быстрее, чем s/[abc]//g.
- Функция print с разделителем-запятой может оказаться быстрее конкатенации строк. Например, код:

```
print $fullname{$name} . " has a new home directory " .
    $home{$name} . "\n";
```

должен склеить вместе два хеша и две фиксированные строки, прежде чем передать их процедурам вывода низкого уровня, тогда как:

```
print $fullname{$name}, " has a new home directory ",
    $home{$name}, "\n";
```

не должен. С другой стороны, в зависимости от значений и архитектуры конкатенация может оказаться быстрее. Проверьте сами.

- `join("", ...)` предпочтительнее, чем серия конкатенированных строк. Множественная конкатенация может привести к копированию строк туда и обратно несколько раз. Оператор `join` позволяет избежать этого.
- `split` с фиксированной строкой обычно быстрее, чем `split` с шаблоном. Поэтому используйте `split(/ /, ...)` вместо `split(/ +/, ...)`, если знаете, что пробел будет только один. Однако шаблоны `/\s+/`, `/^/` и `/ /` специально оптимизированы как особый `split` по пробельному символу.
- Предварительное расширение массива или строки может сберечь время. По мере роста строк и массивов Perl расширяет их, размещая в памяти новый экземпляр с некоторым запасом для роста и копируя прежние значения. Предварительное расширение строки оператором `x` или массива установкой `$#array` могут предотвратить такие эпизодические накладные расходы и уменьшить фрагментацию памяти.
- Не применяйте `undef` к длинным строкам и массивам, если они будут повторно использоваться с теми же целями. Это помогает предотвратить повторное выделение памяти, когда строка или массив должны быть заново расширены.
- Запись `"\0" x 8192` предпочтительнее, чем `unpack("x8192",())`.
- `system("mkdir ...")` может оказаться быстрее для нескольких каталогов, если системный вызов *mkdir* недоступен.
- Избегайте применения функции `eof`, если возвращаемые значения уже указывают на конец файла.
- Кэшируйте записи из файлов (таких как *passwd* и *group*), которые могут многократно использоваться. Особенно важно кэшировать записи, полученные через сеть. Например, для кэширования значений, возвращаемых `gethostbyaddr` при преобразовании числовых адресов (типа `204.148.40.9`) в имена (типа «`www.oreilly.com`»), можно использовать что-то вроде:

```
sub numtoname {
    local ($_) = @_;
    unless (defined $numtoname{$_}) {
        my (@a) = gethostbyaddr(pack('C4', split(/\./)),2);
        $numtoname{$_} = @a > 0 ? $a[0] : $_;
    }

    return $numtoname{$_};
}
```

- Избегайте лишних системных вызовов. Вызовы операционной системы обходятся довольно дорого. Поэтому, например, не вызывайте оператор `time`, если можно обойтись кэшированным значением `$now`. Используйте специальный указатель файла _, чтобы избежать ненужных вызовов *stat*(2). В некоторых системах даже минимальный системный вызов может выполнять тысячу инструкций.
- Избегайте ненужных вызовов `system`. Функция `system` ответвляет подпроцесс, чтобы выполнить заданную вами программу или, что еще хуже, вызвать оболочку для выполнения программы. Это легко может обернуться выполнением миллиона инструкций.
- Остерегайтесь запуска подпроцессов, если это происходит часто. Запуск одного процесса *pwd*, *hostname* или *find* не сильно вам повредит – в конце концов, оболочка занимается запуском подпроцессов постоянно. Можете не верить, но мы иногда поощряем подход ящика с инструментами.
- Сами следите за своим рабочим каталогом, избегая многократных вызовов *pwd*. (Для этого имеется стандартный модуль. См. `Cwd` в главе 30 «Стандартная библиотека Perl».)
- Избегайте использования в командах метасимвола оболочки – где это возможно, передавайте в `system` и `exec` списки.
- Устанавливайте липкий (sticky) бит на интерпретатор Perl на машинах без подкачки по запросу:

```
chmod +t /usr/bin/perl
```

- Для кэширования результатов функций используйте модуль Memoize из CPAN.

Эффективность по памяти

- Для компактного хранения массива целых чисел можно использовать функцию `vec`, если целые числа имеют фиксированную ширину. (Целые числа переменной ширины можно хранить в строке UTF-8.)
- Числовые значения предпочтительнее эквивалентных строковых значений – они занимают меньше памяти.
- Используйте `substr`, чтобы хранить строки постоянной длины в более длинной строке.
- Для очень компактного хранения массива хешей, если длина ключа и значения фиксированы, применяйте модуль `Tie::SubstrHash`.
- Используйте `__END__` и указатель файла `DATA`, чтобы избежать хранения данных программы одновременно в виде строки и массива.
- Применяйте `each` вместо `keys`, если порядок не имеет значения.
- Удаляйте или делайте неопределенными не используемые более глобальные переменные.

- Организуйте хранение хешей в каком-нибудь файле DBM.
- Сохраняйте массивы во временных файлах.
- Используйте каналы для передачи обработки другим инструментам.
- Избегайте списковых операций и загрузки файлов целиком.
- Избегайте `tr///`. Каждое выражение `tr///` должно содержать большую таблицу трансляции.
- Не разворачивайте циклы и старайтесь обойтись без подстановки (inline) подпрограмм.

Для эффективности программирования

- Используйте значения по умолчанию.
- Применяйте традиционные сокращенные ключи командной строки, такие как `-a`, `-n`, `-p`, `-s` и `-i`.
- Используйте `for` в смысле `foreach`.
- Выполняйте системные команды с обратными штрихами.
- Используйте `<*>` и тому подобное.
- Старайтесь работать с шаблонами, создаваемыми на этапе исполнения.
- Щедро используйте в своих шаблонах `*`, `+` и `{}`.
- Обрабатывайте целые массивы и считывайте целые файлы.
- Используйте `getc`.
- Используйте `$\``, `$&` и `$'`.
- Не проверяйте значения ошибок в `open`, т. к. `<`*`HANDLE`*`>` и `print` *`HANDLE`* ведут себя просто как пустые операции, если им передан неверный указатель.
- Не вызывайте `close` для своих файлов – они будут закрыты при очередном `open`.
- Не передавайте подпрограммам аргументы. Используйте глобальные переменные.
- Не давайте имен параметрам подпрограмм. К ним можно обращаться непосредственно как к `$_[`*`EXPR`*`]`.
- Используйте первое, что придет в голову.

Для эффективности сопровождения

- Не используйте значения по умолчанию.
- Используйте `foreach` в смысле `foreach`.
- Используйте осмысленные метки циклов в `next` и `last`.
- Давайте осмысленные имена переменным.
- Давайте осмысленные имена подпрограммам.

- Ставьте в начало строки то, что важно, используя `and`, `or` и модификаторы команд (типа `exit if $done`).
- Закрывайте свои файлы, закончив с ними работу.
- Используйте пакеты, модули и классы, чтобы скрыть детали реализации.
- Передавайте аргументы как параметры процедурам.
- Именуйте параметры процедур с помощью `my`.
- Расставляйте круглые скобки для ясности.
- Расставляйте побольше (полезных) комментариев.
- Включайте встроенную pod-документацию.
- `use warnings`.
- `use strict`.

Для эффективности перенесения

- Помашите приличными чаевыми перед носом носильщика.
- Избегайте функций, которые не везде реализованы. Проверить, какие из них доступны, можно при помощи операторов `eval`.
- Используйте модуль `Config` или переменную `$^O`, чтобы узнать тип машины, на которой вы работаете.
- Не рассчитывайте, что на неизвестной машине будут успешно выполняться `pack` и `unpack` для чисел с плавающей точкой в родном для нее формате.
- Используйте сетевой порядок байтов (форматы `"n"` и `"N"` для `pack`) при отправке двоичных данных по сети.
- Не посылайте двоичные данные по сети. Посылайте ASCII. Лучше посылать UTF-8. Еще лучше посылать деньги.
- Проверяйте `$]` или `$^V`, чтобы узнать, поддерживает ли текущая версия все используемые вами возможности.
- Избегайте применения `$]` и `$^V`. Используйте `require` или `use` с номером версии.
- Вставляйте «хак» `eval exec`, даже если вы не используете его, чтобы ваша программа работала на тех немногих системах, в которых есть Unix-подобные оболочки, но которые не распознают обозначение `#!`.
- Вставляйте строку `#!/usr/bin/perl`, даже если вы не используете ее.
- Проверяйте варианты команд Unix. Некоторые программы *find*, например, не могут обрабатывать ключ *-xdev*.
- Избегайте различных команд Unix, которые можно выполнить внутренне. Команды Unix не слишком хорошо работают в MS-DOS или VMS.
- Помещайте все свои сценарии и страницы руководства в одну сетевую файловую систему, которая смонтирована на всех ваших машинах.

- Опубликуйте свой модуль в CPAN. Вы получите массу откликов, если он окажется не переносимым на другие платформы.

Для эффективности использования

- Вместо того чтобы заставлять ваших пользователей вводить строку за строкой, переведите их в любимый ими редактор.
- Еще лучше, используйте GUI типа расширения Perl/Tk, в котором пользователи могут контролировать порядок событий. (Perl/Tk имеется в CPAN.)
- Дайте пользователям возможность что-нибудь почитать, пока вы продолжаете работу.
- Используйте автозагрузку, чтобы *казалось*, будто программа работает быстро.
- Предоставьте возможность вывода полезных сообщений для каждого приглашения.
- Выведите полезное сообщение о правилах работы, если пользователь ввел некорректные данные.
- Предлагайте значения по умолчанию для каждого приглашения и, возможно, несколько вариантов.
- Предусмотрите значения по умолчанию, предназначенные для новичков. Разрешите опытным пользователям изменять значения по умолчанию.
- Используйте для ввода один символ, если это имеет смысл.
- Организуйте взаимодействие с пользователем по образцам, с которыми он (пользователь) знаком.
- Сделайте так, чтобы из сообщения об ошибке было понятно, что именно требуется исправить. Включайте всю относящуюся к делу информацию, в том числе имя файла и код ошибки, например:

```
open(FILE, $file) or die "$0: Невозможно открыть $file для чтения: $!\n";
```

- Используйте `fork && exit`, чтобы отключиться (отсоединиться) от терминала, если оставшаяся часть вашего сценария является просто пакетной обработкой.
- Разрешите аргументам поступать из командной строки или со стандартного устройства ввода.
- Не устанавливайте произвольных ограничений в своей программе.
- Используйте поля переменной длины, а не фиксированной.
- Используйте сетевые протоколы, ориентированные на текст.
- Посоветуйте всем остальным использовать сетевые протоколы, ориентированные на текст!

- Посоветуйте всем остальным советовать всем остальным использовать сетевые протоколы, ориентированные на текст!!!
- Будьте ленивы во благо других.
- Будьте вежливы.

Стиль программирования

У каждого, конечно, есть свои предпочтения в отношении форматирования, но существуют некоторые общие руководящие указания, которые позволяют облегчить чтение, понимание и сопровождение программ.

Самое важное – запускать программы с директивой `use warnings`. (Нежелательные переменные можно отключить с помощью `no warnings`.) Следует также всегда выполнять программу с директивой `use strict`, если нет достаточных оснований, чтобы обойтись без нее. Полезными могут также оказаться директивы `use sigtrap` и даже `use diagnostics`.

В отношении эстетики оформления кода едва ли не единственное, о чем сильно беспокоится Ларри, это чтобы закрывающая фигурная скобка многострочного блока имела выступ и была выровнена по ключевому слову, которое начало конструкцию. Помимо того, у него есть другие предпочтения, которые не столь сильны. Примерам данной книги (должны быть) свойственны следующие соглашения по кодированию:

- Использование отступов в четыре колонки (на размер табуляции).
- Открывающая фигурная скобка должна находиться, по возможности, в той же строке, что и предшествующее кодовое слово; в противном случае их следует выравнивать вертикально.

```
while ($condition) {        # для коротких выравнивание с ключевыми словами
    # какие-то действия
}

# если условие переносится, выравнивать фигурные скобки друг с другом
while ($this_condition and $that_condition
       and $this_other_long_condition)
{
    # какие-то действия
}
```

- Ставьте пробел перед открывающей фигурной скобкой многострочного блока.
- Короткий блок можно поместить в одной строке вместе с фигурными скобками.
- Опустите точку с запятой в коротком однострочном блоке *BLOCK*.
- Возьмите за правило окружать операторы пробелами.
- Окружайте «сложный» индекс (внутри квадратных скобок) пробелами.

- Помещайте пустые строки между фрагментами кода, выполняющими разные функции.
- Разделяйте закрывающую фигурную скобку и `else` посредством перевода строки.
- Не ставьте пробел между именем функции и открывающей круглой скобкой.
- Не ставьте пробел перед точкой с запятой.
- Ставьте пробел после каждой запятой.
- Переносите длинные строки после оператора (но перед `and` или `or`, даже в виде `&&` и `||`).
- Выравнивайте соответствующие элементы по вертикали.
- Опускайте избыточные символы пунктуации, если при этом не страдает ясность.

У Ларри есть обоснования для каждого из этих пристрастий, но он не утверждает, что у всех остальных голова работает (или не работает) так же, как у него.

Вот несколько еще более существенных соображений относительно стиля:

- Если нечто *может быть* сделано определенным образом, это еще не значит, что вы *должны* сделать это именно так. Perl разработан так, чтобы обеспечить несколько способов сделать что-то, поэтому выберите тот, который лучше всего читается. Например:

  ```
  open(FOO,$foo)  or die "Can't open $foo: $!";
  ```

  лучше чем:

  ```
  die "Can't open $foo: $!"   unless open(FOO,$foo);
  ```

  потому что второй способ скрывает основной смысл команды в модификаторе. Опять же,

  ```
  print "Starting analysis\n" if $verbose;
  ```

  лучше, чем

  ```
  $verbose and print "Starting analysis\n";
  ```

  поскольку главное не в том, ввел пользователь `-v` или нет.
- Аналогично, если оператор допускает аргументы по умолчанию, это не значит, что вы *должны* использовать значения по умолчанию. Они существуют для ленивых программистов, которые пишут программы «одним махом». Если вы хотите, чтобы вашу программу можно было читать, снабжайте ее аргументами.
- Аналогично, то обстоятельство, что во многих местах можно опускать круглые скобки, не означает, что это делать обязательно:

```
return print reverse sort num values %array;
return print(reverse(sort num (values(%array))));
```

Если есть сомнения, ставьте круглые скобки. По крайней мере, они позволят какому-нибудь несчастному понажимать клавишу % в *vi*.

Даже если *у вас* нет сомнений, подумайте о том, кому придется сопровождать код после вас и кто может поставить круглые скобки в неверном месте.

- Не следует прибегать к специальным уловкам, чтобы выйти из цикла через начало или конец. Perl предоставляет оператор `last`, обеспечивающий возможность выхода в середине. Его можно сделать более заметным с помощью выступа:

```
LINE:
    for (;;) {
        statements;
      last LINE if $foo;
        next LINE if /^#/;
        statements;
    }
```

- Не бойтесь использовать метки циклов – они служат улучшению читаемости, а также позволяют предусмотреть несколько уровней для выхода из цикла. См. только что приведенный пример.
- Избегайте применения `grep`, `map` и обратных штрихов в пустом контексте, т. е. если возвращаемые ими значения просто отбрасываются. У всех этих функций есть возвращаемые значения, поэтому используйте их. В противном случае следует предпочесть цикл `foreach` или функцию `system`.
- Для лучшей переносимости при использовании функций, которые могут не быть реализованы на любой машине, проверьте конструкцию в `eval`, чтобы убедиться, является ли вызов успешным. Зная номер версии или уровень патча для какой-то функции, можно проверить `$]` (`$PERL_VERSION` в модуле English), чтобы узнать, присутствует ли эта функция. Модуль `Config` также позволяет запросить значения, которые определила программа *Configure* при установке Perl.
- Тщательнее выбирайте мнемоники для идентификаторов. Если вы не можете вспомнить, что означает мнемоника, начинаются трудности.
- Хотя короткие идентификаторы типа `$gotit` не вызывают возражений, для разделения слов лучше использовать символ подчеркивания. Как правило, значительно легче прочесть `$var_names_like_this`, чем `$VarNamesLikeThis`, особенно тем, для кого английский не является родным языком. Кроме того, такое же правило относится к `$VAR_NAMES_LIKE_THIS`.

  Имена пакетов иногда являются исключениями из этого правила. Perl неформально резервирует имена модулей на нижнем регистре для модулей директив, таких как `integer` и `strict`. Имена других модулей должны начинаться с заглавной буквы и включать символы разных регистров,

но, возможно, не должны включать символы подчеркивания из-за ограничений длины имен в примитивных файловых системах.

- Может оказаться целесообразным использовать регистр букв для указания области видимости или природы переменных. Например:

```
$ALL_CAPS_HERE   # только константы (остерегайтесь конфликтов с переменными
                   Perl!)
$Some_Caps_Here  # глобальные/статические переменные пакета
$no_caps_here    # переменные области видимости функции my() или local()
```

По различным неясным причинам имена функций и методов лучше всего действуют, когда они полностью записаны на нижнем регистре. Например, `$obj->as_string()`.

Посредством ведущего символа подчеркивания можно указать, что переменная или функция не должны использоваться вне пакета, в котором они определены. (Perl не требует соблюдения этого правила, это просто форма документирования.)

- Если вам попалось действительно трудное регулярное выражение, используйте модификатор `/x` и добавьте пробельные символы, чтобы оно было меньше похоже на случайный шум.
- Избегайте применения косой черты в качестве разделителя, если в вашем регулярном выражении уже слишком много прямых и обратных слэшей.
- Не используйте кавычки в качестве разделителя, если в строке уже есть какие-либо кавычки. Обратитесь к псевдофункциям `q//`, `qq//` или `qx//`.
- Операторы `and` и `or` избавляют от необходимости заключать в круглые скобки списковые операторы и позволяют отказаться от избыточных операторов пунктуации типа `&&` и `||`. Вызывайте свои подпрограммы, как если бы они были функциями или списковыми операторами, чтобы избежать лишних амперсандов и круглых скобок.
- Используйте встроенные документы вместо повторения команд `print`.
- Выравнивайте однотипные элементы по вертикали, особенно если они слишком длинны и не умещаются в одной строке:

```
$IDX = $ST_MTIME;
$IDX = $ST_ATIME       if $opt_u;
$IDX = $ST_CTIME       if $opt_c;
$IDX = $ST_SIZE        if $opt_s;

mkdir $tmpdir, 0700 or die "can't mkdir $tmpdir: $!";
chdir($tmpdir)      or die "can't chdir $tmpdir: $!";
mkdir 'tmp',   0777 or die "can't mkdir $tmpdir/tmp: $!";
```

- Истина, которую мы повторим трижды:

  Обязательно проверяйте значения, возвращаемые системными вызовами.

  Обязательно проверяйте значения, возвращаемые системными вызовами.

ОБЯЗАТЕЛЬНО ПРОВЕРЯЙТЕ ЗНАЧЕНИЯ, ВОЗВРАЩАЕМЫЕ СИСТЕМНЫМИ ВЫЗОВАМИ.

Сообщения об ошибках должны выводиться на STDERR и должны указывать, какая программа создала проблему и каким был системный вызов и его аргументы. Важно, чтобы для неудачных системных вызовов сообщения содержали стандартные системные сообщения об ошибках. Вот простой, но показательный пример:

```
opendir(D, $dir)  or die "Невозможно выполнить opendir $dir: $!";
```

- Выравнивайте транслитерации, когда это имеет смысл:

```
tr [abc]
   [xyz];
```

- Думайте о повторном использовании кода. Зачем тратить умственные усилия на одноразовый сценарий, если что-нибудь подобное может понадобиться потом снова? Попробуйте обобщить свой код. Рассмотрите возможность создания модуля или класса объектов. Заставьте свой код чисто выполняться при действии use strict и -w. Рассмотрите возможность бесплатного распространения своего кода. Попробуйте изменить свой взгляд на мир. Попробуйте... а-а, не обращайте внимание.
- Будьте последовательны.
- Будьте вежливы.

Свободный разговор на Perl

Мы коснулись нескольких идиом в предшествующих разделах (не говоря уже о предыдущих главах), но есть много других идиом, которые вы часто встретите, читая программы опытных программистов Perl. Говоря об идиоматическом Perl в данном контексте, мы имеем в виду не набор произвольных выражений Perl с окаменевшими значениями. Скорее мы имеем в виду код Perl, который показывает понимание общего строя языка, что и когда вам может сойти с рук и что вам это даст.

Мы не рассчитываем перечислить все идиомы, которые могут вам встретиться, – для этого потребовалась бы еще одна такая же большая книга. Или две. См. например «Perl Cookbook», Tom Christiansen and Nathan Torkington, O'Reilly&Associates («Perl: библиотека программиста», Т. Кристиансен и Н. Торкингтон, издательство «Питер», 2000). Но вот несколько важных идиом, где «важный» можно определить как «то, что вызывает свист и шиканье тех, кто считает, что им известно, какими должны быть языки программирования».

- Используйте => вместо запятой, если это улучшает читаемость:

```
return bless $mess => $class;
```

Что означает: «Освятить эту мешанину в указанный класс». Остерегайтесь применять => после слова, которое не должно автоматически заключаться в кавычки:

```
sub foo () { "FOO" }
sub bar () { "BAR" }
print foo => bar;   # выводит fooBAR, а не FOOBAR;
```

Хорошо также использовать => рядом с литеральной запятой, разглядеть которую может быть нелегко:

```
join(", " => @array);
```

Perl предоставляет возможность сделать что-то несколькими способами, чтобы вы могли проявить свои творческие возможности. Проявляйте их!

- Используйте местоимение в единственном числе для улучшения читаемости:

```
for (@lines) {
    $_ .= "\n";
}
```

Переменная $_ является вариантом Perl для личного местоимения. Поэтому приведенный код означает «возьмите каждую строку и прикрепите к *ней* символ перевода строки». В современном языке можно даже записать это:

```
$_ .= "\n" for @lines;
```

Местоимение $_ настолько важно для Perl, что его употребление обязательно в grep и map. Вот способ организации кэша часто получаемых результатов функции expensive:

```
%cache = map { $_ => expensive($_) } @common_args;
$xval = $cache{$x} || expensive($x);
```

- Для еще лучшей читаемости опустите и местоимение ($_).[1]
- Используйте операторы управления циклами, модифицируемые командами.

```
while (<>) {
    next if /^=for\s+(index|later)/;
    $chars += length;
    $words += split;
    $lines += y/\n//;
}
```

[1] В данном разделе несколько элементов маркированного списка относятся к одному последующему примеру, поскольку некоторые наши примеры иллюстрируют несколько идиом.

Это фрагмент кода, посредством которого мы подсчитывали страницы в данной книге. Когда приходится выполнять много работы с одной и той же переменной, часто, вопреки распространенному мнению, лучше читается код вообще без местоимений.

Этот фрагмент демонстрирует также идиоматическое использование `next`, модифицируемого командой для закорачивания цикла.

Переменная `$_` всегда является управляющей для цикла в `grep` и `map`, но ссылка программы на нее часто неявная:

```
@haslen = grep { length } @random;
```

Здесь мы берем список скаляров и выбираем из них только те, длина которых больше нуля.

- Используйте `for` для установки антецедента местоимения:

```
for ($episode) {
    s/fred/barney/g;
    s/wilma/betty/g;
    s/pebbles/bambam/g;
}
```

Что из того, что в цикле только один элемент? А то, что это удобный способ «настроить» местоимение, т. е. `$_`. В лингвистике это называется установкой темы (topicalization) и служит не обману, а коммуникации.

- Неявно ссылайтесь на местоимение множественного числа, `@_`.
- Используйте операторы управления для установки значений по умолчанию:

```
sub bark {
    my Dog $spot = shift;
    my $quality  = shift || "yapping";
    my $quantity = shift || "nonstop";
     ...
}
```

Здесь мы неявно используем другое местоимение Perl, `@_`, которое означает «они, их» (them). Аргументы функции всегда приходят как «они». Оператор `shift` знает, что нужно выполнить действие с содержимым `@_`, если она опущена.

Хотя «по происхождению» `||` – логический оператор, с его помощью можно устанавливать значения по умолчанию, поскольку Perl возвращает первое истинное значение. Программисты Perl часто бесцеремонно обращаются с истиной; приведенная строка не дала бы желаемого, если бы вы, например, задали `quantity` равным нулю. Но если вы не собираетесь устанавливать `$quality` или `$quantity` в значение «ложь», идиома работает прекрасно. Нет смысла впадать в суеверие и расставлять повсюду `defined` и `exists`. Просто нужно понять, что она делает. Пока случайно не будет получено ложное значение, все хорошо.

- Используйте формат операторов с присваиванием, в том числе для операторов управления:

```
$xval = $cache{$x} ||= expensive($x);
```

Здесь мы вовсе не инициализируем наш кэш. Мы просто полагаемся на то, что оператор `||=` вызовет `expensive($x)` и присвоит результат `$cache{$x}`, только если `$cache{$x}` имеет ложное значение. Результатом будет то новое значение, которое получит `$cache{$x}`. Снова мы бесцеремонно обращаемся с истиной, поскольку если мы поместим в кэш ложное значение, то `expensive($x)` будет вызвана снова. Может быть, программист знает, что это нормально, поскольку вызов `expensive($x)` недорого стоит, когда она возвращает ложь. А может быть, программист знает, что `expensive($x)` вообще не может возвратить ложное значение. А может быть, программист просто неряшлив. Неряшливость можно рассматривать как форму творчества.

- Используйте операторы управления циклом как операторы, а не просто команды. И...
- Используйте запятые как «маленькие» точки с запятой в небольшом блоке:

```
while (<>) {
    $comments++, next if /^#/;
    $blank++, next    if /^\s*$/;
    last              if /^__END__/;
    $code++;
}
print "comment = $comments\nblank = $blank\ncode = $code\n";
```

Здесь проявлено понимание того, что модификаторы команд модифицируют команды, а `next` является просто оператором. Здесь показано также идиоматическое использование запятой для разделения выражений, во многом сходное с обычным употреблением точки с запятой. (Различие в том, что запятая удерживает два выражения в составе одной команды под контролем одного модификатора команды.)

- Используйте к своей выгоде управление последовательностью выполняемых команд:

```
while (<>) {
    /^#/       and $comments++, next;
    /^\s*$/    and $blank++, next;
    /^__END__/ and last;
    $code++;
}
print "comment = $comments\nblank = $blank\ncode = $code\n";
```

Это снова тот же самый цикл, на этот раз с круглыми скобками в начале. Проницательный программист Perl поймет, что он (этот цикл) компилируется в те же самые внутренние коды, что и предыдущий пример. Моди-

фикатор `if` является просто обратной конъюнкцией `and` (или `&&`), а модификатор `unless` – обратной дизъюнкцией `or` (или `||`).

- Используйте неявные циклы, создаваемые ключами `-n` и `-p`.
- Не ставьте точку с запятой в конце однострочного блока:

```
#!/usr/bin/perl -n
$comments++, next LINE if /#/;
$blank++, next LINE    if /^\s*$/;
last LINE              if /^__END__/;
$code++;

END { print "comment = $comments\nblank = $blank\ncode = $code\n" }
```

Это, в сущности, та же программа, что и раньше. Мы поместили метку `LINE` на операторах управления циклом по собственному желанию, а не из-за реальной необходимости, т. к. неявный цикл `LINE`, создаваемый ключом `-n`, является самым внутренним охватывающим циклом.

Мы использовали `END`, чтобы вывести завершающую команду за пределы неявного основного цикла, так же как в *awk*.

- Применяйте встроенные документы, когда печать становится устрашающей.
- Используйте осмысленный разделитель для встроенного документа:

```
END { print <<"COUNTS" }
comment = $comments
blank = $blank
code = $code
COUNTS
```

Вместо нескольких команд вывода программист, свободно пишущий на Perl, использует строку, располагаемую на нескольких строчках, с интерполяцией. И несмотря на то что ранее мы назвали это распространенной ошибкой, здесь мы без зазрения совести пропустили завершающую точку с запятой, потому что она не обязательна в конце блока `END`. (Если когда-нибудь мы преобразуем его в многострочный блок, то вернем точку с запятой обратно.)

- Выполняйте подстановки и трансляции со скалярами на проходе:

```
($new = $old) =~ s/bad/good/g;
```

Поскольку l-значениям можно присваивать значения, вы часто увидите, как значение меняется «на проходе» во время присваивания. Это может на практике предотвратить внутреннее копирование (если мы когда-нибудь соберемся реализовать соответствующую оптимизацию):

```
chomp($answer = <STDIN>);
```

Любая функция, модифицирующая аргумент по месту, может использоваться в приеме «на проходе». Но погодите, это еще не все!

- Не ограничивайтесь изменением скаляров на проходе:

```
for (@new = @old) { s/bad/good/g }
```

  Здесь мы копируем @old в @new, изменяя все на ходу (не все сразу, конечно, – блок выполняется многократно, каждый раз с новым значением).

- Передавайте именованные параметры с помощью фигурного оператора запятой =>.
- Предоставьте хешу при присваивании осуществлять обработку четных/нечетных аргументов:

```
sub bark {
    my DOG $spot = shift;
    my %parm = @_;
    my $quality  = $parm{QUALITY}  || "yapping";
    my $quantity = $parm{QUANTITY} || "nonstop";
     ...
}

$fido->bark( QUANTITY => "once",
             QUALITY => "woof" );
```

  Именованные параметры часто являются роскошью (которую можно себе позволить). А в Perl вы получаете их даром, если не считать стоимость присваивания хешу.

- Повторяйте булевы выражения до получения ложного значения.
- Используйте минимальное соответствие, если это возможно.
- Используйте модификатор /e для вычисления выражения замены:

```
#!/usr/bin/perl -p
1 while s/^(.*?)(\t+)/$1 . ' ' x (length($2) * 4 - length($1) % 4)/e;
```

  Эта программа исправляет файлы, получаемые от того, кто ошибочно полагает, что может переопределить аппаратную табуляцию так, чтобы она занимала 4 пробела, а не 8. В ней используется несколько важных идиом. Во-первых, идиома 1 while удобна, когда все, что вы хотите делать в цикле, фактически осуществляется в условии. (Perl достаточно сообразителен, чтобы не предупреждать об использовании 1 в пустом контексте.) Необходимо повторять эту подстановку, поскольку каждый раз, заменяя несколько символов табуляции пробелами, приходится пересчитывать позицию колонки следующей табуляции от начала.

  (.*?)ищет как можно меньшую строку до первой табуляции, используя модификатор минимального соответствия (знак вопроса). В данном случае мы могли прибегнуть к обычному жадному поиску * следующим образом: ([^\t]*). Но это срабатывает только потому, что табуляция является одиночным символом, благодаря чему можно использовать отрицание класса символов, чтобы не «проскочить» мимо первой табуляции. В целом, минимальный поиск значительно элегантнее и останется работоспо-

собным, даже если следующее отыскиваемое соответствие окажется длиннее одного символа.

Модификатор /e осуществляет подстановку, используя выражение, а не просто строку. Это позволяет выполнять необходимые вычисления именно тогда, когда это требуется.

- Сложные подстановки требуют творческого подхода к форматированию и комментариев:

```
#!/usr/bin/perl -p
1 while s{
    ^               # привязка к началу
    (               # начало первой подгруппы
        .*?         # поиск минимального числа символов
    )               # конец первой подгруппы
    (               # начало второй подгруппы
        \t+         # поиск одной или более табуляций
    )               # конец второй подгруппы
}
{
    my $spacelen = length($2) * 4;  # учесть полные табуляции
    $spacelen -= length($1) % 4;    # учесть неполные табуляции
    $1 . ' ' x $spacelen;           # задать правильное число пробелов
}ex;
```

Это, возможно, излишне, но некоторых впечатляет больше, чем предыдущий однострочник. Решайте сами.

- Не бойтесь применять $`, если вам того хочется:

```
1 while s/(\t+)/' ' x (length($1) * 4 - length($`) % 4)/e;
```

Вот более короткая версия, включающая $`, о которой известно, что она ухудшает производительность. Но мы используем только ее длину, поэтому не считается, что это плохо.

- Используйте смещения непосредственно из массивов @- (@LAST_MATCH_START) и @+ (@LAST_MATCH_END):

```
1 while s/\t+/' ' x (($+[0] - $-[0]) * 4 - $-[0] % 4)/e;
```

Это даже короче. (Если вы не видите здесь массивов, попробуйте найти элементы массивов.) См. @- и @+ в главе 28.

- Используйте eval с возвращаемой константой:

```
sub is_valid_pattern {
    my $pat = shift;
    return eval { "" =~ /$pat/; 1 } || 0;
}
```

Совершенно необязательно, чтобы оператор eval {} возвращал реальное значение. Здесь мы всегда возвращаем 1, если он выполняется до конца.

Однако если шаблон в $pat окажется некорректным, eval перехватит это и возвратит undef в логическое условие оператора ||, который превратит его в определенный 0 (из вежливости, поскольку undef – тоже ложное значение, но из-за него кто-то может подумать, что глючит подпрограмма is_valid_pattern, а нам бы этого не хотелось).

- Выполняйте всю грязную работу при помощи модулей.
- Используйте фабрики объектов.
- Применяйте обратные вызовы.
- Используйте стеки для слежения за контекстом.
- Организуйте доступ к концу массива или строки посредством отрицательных индексов:

```
use XML::Parser;

$p = new XML::Parser Style => 'subs';
setHandlers $p Char => sub { $out[-1] .= $_[1] };

push @out, "";

sub literal {
    $out[-1] .= "C<";
    push @out, "";
}

sub literal_ {
    my $text = pop @out;
    $out[-1] .= $text . ">";
}
...
```

Это отрывок из программы в 250 строк, с помощью которой мы преобразовали XML-версию старой книги Camel обратно в формат pod, чтобы редактировать ее для этого издания в Real Text Editor.

Во-первых, можно заметить, что мы использовали модуль XML::Parser (из CPAN) для правильного разбора нашего XML, поэтому нам не пришлось думать, как это сделать. Это сразу сократило нашу программу на несколько тысяч строк (в предположении, что мы заново реализовали бы на Perl все, что модуль XML::Parser делает вместо нас,[1] в том числе трансляцию почти из любого набора символов в UTF-8).

XML::Parser использует идиому высокого уровня, называемую *фабрикой объектов* (*object factory*). В данном случае это фабрика парсеров (синтаксических анализаторов). Создавая объект XML::Parser, мы сообщаем, какой стиль интерфейса анализатора нам требуется, и он создается для нас. Это отличный способ создать приложение для тестирования, когда не ясно, какой тип интерфейса окажется лучшим в конечном итоге. Стиль subs

[1] На самом деле XML::Parser служит просто красивой оболочкой вокруг XML-анализатора *expat* Джеймса Кларка.

является лишь одним из интерфейсов `XML::Parser`. На самом деле это один из самых старых интерфейсов и, возможно, даже не самый популярный в настоящее время.

В строке `setHandlers` показан вызов метода анализатора не в указательной нотации, а в нотации «косвенного объекта», которая помимо прочего позволяет опустить круглые скобки для аргументов. В строке также использована идиома именованного параметра, которую мы видели выше.

В строке показана также другая мощная концепция – обратный вызов (callback). Вместо того чтобы вызывать анализатор для получения следующего элемента, мы говорим, чтобы он вызывал нас. Для именованных тегов XML, таких как `<literal>`, этот стиль интерфейса автоматически вызывает подпрограмму с таким же именем (или именем с символом подчеркивания на конце для соответствующего закрывающего тега). Но данные между тегами не именованы, поэтому мы устанавливаем обратный вызов `Char` с помощью метода `setHandlers`.

Затем мы инициализируем массив `@out`, который играет роль стека для вывода. Мы помещаем в него нулевую строку, которая означает, что мы не получили никакого текста на данном уровне вложенности тегов (изначально 0).

Именно теперь снова вступает в дело наш обратный вызов. Как только мы видим текст, он автоматически дописывается в последний элемент массива через идиому `$out[-1]` в обратном вызове. На уровне внешнего тега `$out[-1]` является тем же, что и `$out[0]`, поэтому `$out[0]` содержит всю выводимую информацию. (В конечном итоге. Но сначала надо разобраться с тегами.)

Допустим, в тексте встретился тег `<literal>`. Тогда вызывается подпрограмма `literal`, которая дописывает некоторый текст в текущий элемент `@out`, затем проталкивает новый контекст на стек `@out`. Теперь любой текст до закрывающего тега добавляется к этому новому концу стека. Наткнувшись на закрывающий тег, мы выталкиваем собранный нами `$text` обратно из стека `@out` и дописываем оставшуюся часть преобразованных данных к новому (т. е. старому) концу стека, результатом чего будет трансляция строки XML `<literal>text</literal>` в соответствующую строку pod, `C<text>`.

Подпрограммы для других тегов точно такие же, только другие.

- Используйте `my` без присваивания для создания пустого массива или хеша.
- Расщепляйте строку по умолчанию по пробельным символам.
- Выполняйте присваивание спискам переменных, чтобы собрать любое нужное вам количество.
- Используйте самооживление неопределенных ссылок для их создания.
- Автоинкрементируйте неопределенные элементы массивов и хешей, чтобы создать их.

- Используйте автоинкрементрование хеша `%seen` для определения уникальности.
- Делайте присваивание удобной временной переменной `my` в условном операторе.
- Используйте режим автоматической расстановки кавычек в скобках.
- Применяйте альтернативный механизм расстановки кавычек для вставки двойных кавычек.
- Используйте оператор `?:` для переключения между двумя аргументами в `printf`.
- Выравнивайте аргументы `printf` по их полям `%`:

```
my %seen;
while (<>) {
    my ($a, $b, $c, $d) = split;
    print unless $seen{$a}{$b}{$c}{$d}++;
}
if (my $tmp = $seen{fee}{fie}{foe}{foo}) {
    printf qq(Saw "fee fie foe foo" [sic] %d time%s.\n"),
                                        $tmp,  $tmp == 1 ? "" : "s";
}
```

Эти девять строк битком набиты идиомами. Первая строка создает пустой хеш, потому что мы ничего ему не присваиваем. Мы производим итерацию по входным строкам, неявно устанавливая `$_`, затем используем `split` без аргументов, которая расщепляет ее по пробельным символам. Затем выбираем первые четыре слова с помощью присваивания списку и отбрасываем все остальные. Затем мы запоминаем первые четыре слова в четырехмерном хеше, что автоматически создает (при необходимости) первые три элемента ссылок и последний элемент счетчика, который инкрементируется оператором автоинкрементирования. (При действии `use warnings` автоинкрементирование не выдает предупреждений об использовании неопределенных значений, поскольку автоинкрементирование является допустимым способом определения неопределенных значений.) Затем мы выводим строку, если ранее не встречалась строка, начинающаяся с этих четырех слов, потому что автоинкрементирование является постинкрементированием, которое помимо увеличения значения хеша возвращает прежнее значение, если оно имелось.

По окончании цикла мы снова проверяем `%seen`, чтобы узнать, встретилась ли конкретная комбинация четырех слов. Мы используем то обстоятельство, что литеральный идентификатор можно поместить в фигурные скобки, и он автоматически будет заключен в кавычки. В противном случае нам бы пришлось сказать `$seen{"fee"}{"fie"}{"foe"}{"foo"}`, что противно, даже если вы не убегаете от великана.

Мы присваиваем результат `$seen{fee}{fie}{foe}{foo}` временной переменной до проверки его в булевом контексте, предоставляемом `if`. Поскольку

присваивание возвращает свое левое значение, мы все же можем проверить истинность значения. Увидев `my`, вы должны были бы понять, что это новая переменная, и мы не проверяем равенство, а выполняем присваивание. Все прекрасно работало бы и без `my`, а опытный программист Perl все равно сразу заметил бы, что мы использовали один символ `=` вместо двух `==`. (Малоопытный программист Perl мог, однако, быть введен в заблуждение. Программисты Pascal любого уровня мастерства будут взбешены.)

Переходя к команде `printf`, можно видеть, что используется форма двойных кавычек `qq()`, что дает возможность вставить обычные двойные кавычки и символ перевода строки. Мы могли бы прямо интерполировать туда переменную `$tmp`, поскольку это, по существу, строка в двойных кавычках, но мы предпочли дальнейшую интерполяцию через `printf`. Наша временная переменная `$tmp` теперь вполне удобна, особенно потому, что мы хотим не просто интерполировать ее, но также проверить в условии оператора `?:`, чтобы узнать, нужно ли использовать слово «time» во множественном числе. Наконец, обратите внимание, что мы выровняли два поля по соответствующим им маркерам `%` в формате `printf`. Если аргумент слишком длинный, можно перейти на следующую строку для следующего аргумента, чего нам в этом случае делать не пришлось.

Уф! Достаточно? Есть много других идиом, которые мы могли бы обсуждать, но эта книга и так уже достаточно увесиста. Но мы хотим поговорить еще об одном идиоматическом использовании Perl – создании генераторов программ.

Генерирование программ

Как только люди обнаружили, что могут писать программы, они стали писать программы, которые пишут другие программы. Мы часто называем их *генераторами программ*. (Если вы любитель истории, то, возможно, знаете, что RPG означало Report Program Generator – «генератор программ отчетов» – задолго до того, как стало означать Role Playing Game – «ролевая игра».) В наше время их, возможно, назвали бы «фабриками программ», но приверженцы генераторов добрались до них первыми и потому дали им название.

Каждый, кому доводилось писать генератор программ, знает, что от него можно получить косоглазие, даже если быть настороже. Проблема просто в том, что значительная часть данных программы выглядит как настоящий код, но им не является (по крайней мере, до поры). Один и тот же текст содержит то, что делает что-то, и то, что выглядит аналогично, но ничего не делает. В Perl есть различные возможности, которые облегчают смешивание Perl с другими языками, имея в виду текст.

(Конечно, эти возможности облегчают написание Perl на Perl, но мы думаем, что сейчас это скорее из области предположений.)

Генерирование других языков на Perl

Perl (помимо прочего) – это язык обработки текста, а большинство языков программирования являются текстовыми. Кроме того, отсутствие в Perl произвольных ограничений в совокупности с различными механизмами расстановки кавычек и интерполяции облегчают зрительную изоляцию кода другого языка, который вы формируете. Вот, например, небольшой фрагмент *s2p*, транслятора *sed*-в-*perl*:

```
print &q(<<"EOT");
:       #!$bin/perl
:       eval `exec $bin/perl -S \$0 \${1+"\$@"}`
:               if \$running_under_some_shell;
:
EOT
```

Здесь вложенный текст оказывается законным в обоих языках, Perl и *sh*. Мы сразу использовали идиому, которая поможет вам не потерять разум при написании генератора программ: прием с помещением символа «шума» и табуляции в начале каждой цитируемой строки, что зрительно отделяет вложенный код и благодаря чему вы с первого взгляда можете сказать, что это не тот код, который в действительности выполняется. Одна переменная, `$bin`, интерполируется в многострочную цитату в двух местах, а затем строка передается в функцию, которая отделит двоеточие и табуляцию.

Конечно, не обязательно использовать многострочные цитаты. Часто можно увидеть сценарии CGI, содержащие миллионы команд `print`, по одной в каждой строке. Это немного похоже на то, чтобы прибыть в церковь на истребителе, но если он вас туда доставит... (Мы готовы допустить, что колонка команд `print` обладает своей зрительной выразительностью.)

Когда вы встраиваете большую многострочную цитату на каком-нибудь другом языке (например, HTML), часто удобно сделать вид, что вы программируете наоборот, включая Perl в другой язык, во многом подобно тому, как это можно сделать в откровенно вывернутых наизнанку языках типа PHP:

```
print <<"XML";
    <stuff>
    <nonsense>
    blah blah blah @{[ scalar EXPR ]} blah blah blah
    blah blah blah @{[ LIST ]} blah blah blah
    </nonsense>
    </stuff>
XML
```

Вы можете прибегнуть к любому из этих двух приемов для интерполяции выражений произвольной сложности в длинную строку.

Некоторые генераторы программ не очень похожи на генераторы программ, в зависимости от того, какую часть своей работы они скрывают от вас. В главе 22 «CPAN» мы видели, как маленькую программу *Makefile.PL*

можно использовать для написания *Makefile*. Размер *Makefile* может легко оказаться в 100 раз больше, чем породившая его *Makefile.PL*. Подумайте, от какой муки это избавило ваши пальцы. Или не думайте об этом – в этом, в конце концов, и заключается смысл.

Генерирование Perl в других языках

В Perl легко генерировать программы на других языках, но верно и обратное. Код Perl может легко генерироваться в других языках, потому что он сжатый и податливый. Можно выбрать свои кавычки, чтобы они не мешали механизму цитирования другого языка. Не надо беспокоиться об отступах или о том, где поместить переносы строки, или об обратном слэше для обратного слэша. Не обязательно заранее определять пакет как одну строку, потому что разрешается многократно заходить в пространство имен своего пакета, как только вам потребуется выполнить еще код из этого пакета.

Еще одна вещь, которая облегчает написание кода Perl в других языках (включая Perl), – это директива `#line`. Perl умеет обрабатывать ее как специальную директиву, изменяющую его представление об имени текущего файла и номере строки. Это полезно в сообщениях об ошибках и предупреждениях, особенно для строк, обрабатываемых `eval` (которая, если вдуматься, и есть Perl, который пишет Perl). Синтаксис этого механизма тот же, что используется препроцессором C: когда Perl встречает символ `#` и слово `line`, за которыми следуют номер и имя файла, он устанавливает `__LINE__` равным числу, а `__FILE__` равным имени файла.[1]

Вот несколько примеров, которые можно проверить, непосредственно введя в *perl*. Для обозначения конца файла используется <Ctrl>+<D>, что типично для Unix. Пользователи DOS/Windows и VMS могут ввести <Ctrl>+<Z>. Если в вашей оболочке для этого предназначается что-либо иное, вам следует использовать его, чтобы сообщить *perl* об окончании ввода. С другой стороны, всегда можно ввести `__END__`, чтобы сообщить компилятору, что анализировать больше нечего.

Здесь встроенная функция Perl `warn` выводит новые имя файла и номер строки:

```
% perl
# line 2000 "Odyssey"
# the "#" on the previous line must be the first char on line
warn "pod bay doors";  # or die
^D
pod bay doors at Odyssey line 2001.
```

[1] Точнее говоря, он ищет шаблон `/^#\s*line\s+(\d+)\s*(?:\s"([^"]+)")?\s*$/`, при этом `$1` дает номер следующей строки, а `$2` дает необязательное имя файла в кавычках. (Нулевое имя файла оставляет `__FILE__` неизменным.)

А здесь исключительная ситуация, вызываемая `die` в `eval`, попала в переменную `$@` (`$EVAL_ERROR`) вместе с временными новыми именем файла и строкой:

```
# line 1996 "Odyssey"
eval qq{
#line 2025 "Hal"
    die "pod bay doors";
};
print "Problem with $@";
warn "I'm afraid I can't do that";
^D
Problem with pod bay doors at Hal line 2025.
I'm afraid I can't do that at Odyssey line 2001.
```

Это показывает, что директива `#line` действует только на текущую единицу компиляции (файл или `eval` *`STRING`*) и что после окончания компиляции этой единицы автоматически восстанавливаются прежние установки. Благодаря этому вы можете организовывать собственные сообщения внутри `eval` *`STRING`* или `do` *`FILE`*, не затрагивая остальной части своей программы.

Perl поддерживает ключ `-P`, который вызывает препроцессор C, выдающий директивы `#line`. Препроцессор C был первоначальным толчком для реализации `#line`, но в настоящее время он применяется редко, потому что как правило есть более удачные способы осуществления того, для чего предполагалось его использовать. У Perl, однако, есть несколько других препроцессоров, в том числе модуль `AutoSplit`. Препроцессор JPL (Java Perl Lingo) преобразует файлы *.jpl* в файлы *.java*, *.pl*, *.h* и *.c*. Для того чтобы сделать точными сообщения об ошибках, он использует `#line`.

Одним из первых препроцессоров Perl был транслятор *sed*-to-*perl s2p*. На самом деле Ларри отложил первоначальный выпуск Perl, чтобы завершить *s2p* и *awk*-to-*perl* (*a2p*), поскольку полагал, что с ними Perl будет лучше принят. Ну, может быть, они и сыграли свою роль.

Подробнее об этом сказано в электронной документации, а также в трансляторе *find2perl*.

Фильтры исходного кода

Если можно написать программу для трансляции чего угодно в Perl, почему бы не дать возможность вызова этого транслятора из Perl?

Понятие фильтра исходного кода возникло в связи с мыслью, что сценарий или модуль должны уметь расшифровывать себя на лету, например:

```
#!/usr/bin/perl
use MyDecryptFilter;
@*x$]`OuN&k^ZxO2jZ^X{.?s!(f;9Q/^A^@~~8H]|,%@^P:q-=
...
```

Но эта идея получила дальнейшее развитие, и теперь можно определить фильтр исходного кода для осуществления любого преобразования исходного

кода. Соединив это с понятием ключа -x, о котором говорилось в главе 19 «Интерфейс командной строки», можно получить общий механизм, позволяющий загрузить любой программный фрагмент из сообщения и выполнить его, независимо от того, написан он на Perl или нет.

С помощью модуля `Filter` из CPAN сейчас можно делать даже такие вещи, как программировать на Perl в *awk*:

```
#!/usr/bin/perl
use Filter::exec "a2p";          # транслятор awk-to-perl
1,30 { print $1 }
```

Это определенно можно отнести к разряду идиом. Но мы ни в коей мере не будем утверждать, что это часто используемый прием программирования.

25

Переносимость программ Perl

Мир, в котором существует только одна операционная система, делает переносимость легкой, а жизнь скучной. Нам больше по душе обширный генетический пул операционных систем, если только экосистема не разделяется слишком четко на хищников и их добычу. Perl работает на десятках операционных систем, и поскольку программы Perl не являются зависимыми от платформы, то одна и та же программа без модификаций может выполняться на всех этих системах.

Ну, это почти так. Perl старается предоставить программисту как можно больше возможностей, но если использовать особые функции некоторой определенной операционной системы, то это неизбежно уменьшит возможности переноса программы на другие системы. В этом разделе мы дадим некоторые указания относительно написания переносимого кода Perl. Решив, до какой степени вам необходима переносимость, вы будете знать, где лежат границы, и сможете остаться в их пределах.

Если взглянуть на это с другой стороны, то написание переносимого кода обычно означает умышленное ограничение предоставляемых вам возможностей выбора. Естественно, для этого нужны дисциплина и готовность к жертвам, т. е. два качества, обычно несвойственные программистам на Perl.

Имейте в виду, что не все программы Perl должны быть переносимы. Нет оснований использовать Perl в качестве связующего кода для инструментов Unix или для создания прототипов приложений Macintosh, или для управ-

ления реестром Windows. Если есть смысл пожертвовать переносимостью, сделайте это.[1]

В целом заметьте, что понятия ID пользователя, «домашнего» (home) каталога и даже состояния регистрации существуют только на многопользовательских платформах.

Специальная переменная `$^O` сообщает, в какой операционной системе был скомпилирован Perl. Она служит для ускорения кода, которому в противном случае пришлось бы использовать `use Config` для получения тех же данных через `$Config{osname}`. (Даже если `Config` был загружен с другими целями, вы все же сэкономите на издержках поиска в связанном хеше.)

Более подробные сведения о платформе можно получить, просмотрев остальную часть хеша `%Config`, который предоставляется стандартным модулем `Config`. Например, чтобы проверить, поддерживает ли платформа вызов `lstat`, можно посмотреть значение `$Config{d_lstat}`. Полное описание имеющихся переменных можно найти в электронной документации по `Config`, а описание поведения встроенных функций Perl на различных платформах можно найти на странице руководства *perlport*. Вот функции Perl, действие которых больше всего зависит от платформы:

-X (проверки файлов), `accept`, `alarm`, `bind`, `binmode`, `chmod`, `chown`, `chroot`, `connect`, `crypt`, `dbmclose`, `dbmopen`, `dump`, `endgrent`, `endhostent`, `endnetent`, `endprotoent`, `endpwent`, `endservent`, `exec`, `fcntl`, `fileno`, `flock`, `fork`, `getgrent`, `getgrgid`, `getgrnam`, `gethostbyaddr`, `gethostbyname`, `gethostent`, `getlogin`, `getnetbyaddr`, `getnetbyname`, `getnetent`, `getpeername`, `getpgrp`, `getppid`, `getpriority`, `getprotobyname`, `getprotobynumber`, `getprotoent`, `getpwent`, `getpwnam`, `getpwuid`, `getservbyport`, `getservent`, `getservbyname`, `getsockname`, `getsockopt`, `glob`, `ioctl`, `kill`, `link`, `listen`, `lstat`, `msgctl`, `msgget`, `msgrcv`, `msgsnd`, `open`, `pipe`, `qx`, `readlink`, `readpipe`, `recv`, `select`, `semctl`, `semget`, `semop`, `send`, `sethostent`, `setgrent`, `setnetent`, `setpgrp`, `setpriority`, `setprotoent`, `setpwent`, `setservent`, `setsockopt`, `shmctl`, `shmget`, `shmread`, `shmwrite`, `shutdown`, `socket`, `socketpair`, `stat`, `symlink`, `syscall`, `sysopen`, `system`, `times`, `truncate`, `umask`, `utime`, `wait`, `waitpid`

Перевод строки

В большинстве операционных систем строки файлов завершаются одним или двумя символами, уведомляющими о конце строки. Символы эти различны в разных системах. Unix традиционно использует `\012` (т. е. восьмеричный ASCII-код этого символа равен 12), для одного из типов ввода-вывода в DOS предназначена последовательность `\015\012`, а Mac'и используют `\015`. В Perl для представления «логического» перевода строки независимо

[1] Не всякий разговор обязательно корректен в любой культуре. Perl старается дать вам хотя бы один способ сделать Правильное Дело (the Right Thing), но не пытается навязать вам это. В этом отношении Perl более напоминает ваш родной (материнский) язык, а не язык няни.

от платформы служит \n. В MacPerl \n всегда означает \015. В Perl для DOS \n обычно означает \012, но при обращении к файлу в «текстовом режиме» он транслируется в (или из) \015\012, в зависимости от того, читается файл или записывается. Unix делает то же самое на терминалах в каноническом режиме. Обычно \015\012 обозначается как CRLF.

Поскольку DOS различает текстовые и двоичные файлы, то реализации Perl для DOS имеют ограничения при использовании seek и tell с файлами в «текстовом режиме». Для получения лучших результатов выполняйте seek только по позициям, получаемым от tell. Однако в случае применения встроенной функции Perl binmode с указателями файлов обычно от seek и tell никакого вреда не бывает.

Распространенным заблуждением при программировании сокетов является мнение, что \n всюду означает \012. Во многих протоколах Интернета заданы \012 и \015, а значения Perl \n и \r ненадежны, т. к. различаются в зависимости от системы:

```
print SOCKET "Hi there, client!\015\012";  # правильно
print SOCKET "Hi there, client!\r\n";      # неправильно
```

Однако использование \015\012 (или \cM\cJ, или \x0D\x0A, или даже v13.10) может быть утомительным и портить внешний вид, а также смущать тех, кто сопровождает код. Модуль Socket предоставляет кое-какие Правильные Вещи (для тех, кому они нужны):

```
use Socket qw(:DEFAULT :crlf);
print SOCKET "Hi there, client!$CRLF"      # правильно
```

При чтении из сокета помните, что по умолчанию значением разделителя записей $/ является \n, а это значит, что придется проделать некоторую дополнительную работу, если вы не уверены в том, что будете наблюдать в сокете. Надежный код для сокета должен распознавать в качестве конца строки как \012, так и \015\012:

```
use Socket qw(:DEFAULT :crlf);
local ($/) = LF;     # не требуется, если $/ уже \012

while (<SOCKET>) {
    s/$CR?$LF/\n/;   # заменить LF или CRLF логическим переводом строки
}
```

Аналогично, код, возвращающий текстовые данные, – например, подпрограмма для выборки веб-страницы – часто должен производить трансляцию символов перевода строки. Обычно достаточно одной строки кода:

```
$data =~ s/\015?\012/\n/g;
return $data;
```

Остроконечники, тупоконечники и ширина чисел

Компьютеры хранят целые и действительные числа в разном порядке – при этом они делятся на *тупоконечники* (*big-endians*) и *остроконечники* (*little-endians*)[1] – и с разной шириной (32-разрядные и 64-разрядные числа сегодня встречаются чаще всего). Обычно об этом не надо задумываться. Но если программа посылает двоичные данные через сетевое соединение или на диск, который должен читаться другим компьютером, могут потребоваться некоторые меры предосторожности.

Конфликтующие порядки байтов могут превратить числа в полную мешанину. Если остроконечник (например, Intel CPU) записывает число 0×12345678 (десятичное 305 419 896), то машина-тупоконечник (например, с CPU Motorola) прочтет его как 0x78563412 (десятичное 2 018 346). Чтобы избежать этой проблемы при сетевых (через сокеты) соединениях, используйте для `pack` и `unpack` форматы `n` и `N`, которые записывают числа unsigned short и long в порядке *big-endian* (называемом также «сетевым» порядком) независимо от платформы.

Выяснить «конечность» своей платформы можно, распаковав структуру данных, упакованную в родном формате, например:

```
print unpack("h*", pack("s2", 1, 2)), "\n";
# '10002000' на, скажем, Intel x86 или Alpha 21064 в режиме little-endian
# '00100020' на, скажем, Motorola 68040
```

Определить «конечность» можно при помощи любой из следующих команд:

```
$is_big_endian    = unpack("h*", pack("s", 1)) =~ /01/;
$is_little_endian = unpack("h*", pack("s", 1)) =~ /^1/;
```

Даже если две системы характеризуются одинаковой «конечностью», проблемы могут все же возникать при передаче данных между 32-разрядными и 64-разрядными платформами. Единственное подходящее решение – избегать передачи и хранения «сырых» двоичных чисел. Передавайте и храните числа как текст, а не как двоичные данные, или применяйте модули типа `Data::Dumper` и `Storable`, которые сделают это вместо вас. В любом случае лучше использовать протоколы, ориентированные на текст: они более надежны, лучше сопровождаются и расширяются, чем двоичные протоколы.

Конечно, с появлением XML и Unicode определение текста становится более гибким. Например, между двумя системами, на которых выполняется Perl 5.6.0 (или более новая версия), можно передавать последовательность целых чисел, закодированных как символы в `utf8` (Perl-разновидность UTF-8). Если на обоих концах соединения архитектура 64-разрядная, то можно об-

[1] Порядок следования байт в машинном слове зависит от архитектуры, часто определен в системе через псевдопеременную `BYTE_ORDER`. – *Примеч. науч. ред.*

мениваться 64-разрядными целыми числами. В противном случае вы ограничены 32-разрядными целыми. Используйте для отправки `pack` с шаблоном `U*`, а для приема – `unpack` с шаблоном `U*`.

Файлы и файловые системы

Составляющие файлового пути разделяются в Unix символом /, в Windows – символом \, а в Mac'ах – двоеточием (:). Некоторые системы не поддерживают ни жесткие ссылки (`link`), ни символические (`symlink`, `readlink`, `lstat`). Некоторые системы обращают внимание на заглавные буквы в именах, некоторые – нет, а какие-то обращают на это внимание при создании файлов, но не при их чтении.

Существуют модули, способные справиться с этим. Стандартные модули `File::Spec` предоставляют некоторые функции, преследующие цель воплотить Правильную Вещь:

```
use File::Spec::Functions;
chdir( updir() );        # переместиться вверх на один каталог
$file = catfile( curdir(), 'temp', 'file.txt' );
```

Эта последняя строка осуществляет чтение из *./temp/file.txt* в Unix и Windows или *:temp:file.txt* на Mac'ах, или в *[.temp]file.txt* в VMS и запоминает содержимое файла в `$file`.

Модуль `File::Basename` – другой «терпимый» к платформам модуль, поставляемый в связке с Perl, расщепляет имя файла на компоненты: базовое имя файла, полный путь к каталогу и суффикс файла.

Вот несколько советов для создания переносимых программ Perl для манипуляции с файлами:

- Избегайте файлов с именами из одинаковых букв в различных регистрах, например *test.pl* и *Test.pl*, поскольку некоторые платформы игнорируют заглавные буквы.
- По возможности соблюдайте в именах файлов соглашение 8.3 (восьмибуквенные имена и трехбуквенные расширения). Часто можно справиться с более длинными именами, если обеспечить их уникальность при протаскивании через отверстие размером 8.3. (Это должно быть легче, чем протащить верблюда через игольное ушко.)
- Старайтесь, чтобы в именах файлов было как можно меньше символов, не являющихся буквенно-цифровыми. Символы подчеркивания часто разрешены, но при этом теряется символ, который помог бы обеспечить уникальность в системах 8.3. (Помните, что по этой причине мы обычно не включаем символы подчеркивания в имена модулей.)
- Аналогично, применяя модуль `AutoSplit`, постарайтесь ограничить имена подпрограмм восемью или менее символами и не давайте двум подпрограммам имена из одинаковых букв в разных регистрах. Если требуются

более длинные имена подпрограмм, сделайте уникальными первые восемь символов каждого имени.

- Всегда явно указывайте символ < при открытии файла для чтения, иначе в системах, допускающих в именах файлов символы пунктуации, файл, имени которого предшествует символ >, может привести к удалению файла, а файл, имени которого предшествует символ |, может привести к открытию канала. Это вызвано тем, что формат open с двумя аргументами является магическим и интерпретирует такие символы, как >, < и |, что может быть неправильным. (За исключением тех случаев, когда это правильно.)

```
open(FILE,      $existing_file)  or die $!;  # неверно
open(FILE,    "<$existing_file") or die $!;  # правильнее
open(FILE, "<", $existing_file)  or die $!;  # еще правильнее
```

- Не следует предполагать, что текстовые файлы заканчиваются символом перевода строки. Так должно быть, но иногда об этом забывают, особенно если этому помогает текстовый редактор.

Взаимодействие с системой

Платформы, основанные на графическом интерфейсе пользователя, часто не поддерживают командную строку непосредственно, поэтому для обеспечения работоспособности программ, требующих интерфейса командной строки, необходимо предпринять дополнительные шаги.

Некоторые другие советы:

- На некоторых платформах нельзя удалять или переименовывать файлы, которые находятся в работе, поэтому не забывайте закрывать файлы по завершении работы с ними. Не уничтожайте и не переименовывайте открытый файл. Не выполняйте tie или open с уже связанным или открытым файлом; сначала выполните для него untie или close.
- Не открывайте одновременно один и тот же файл для записи более одного раза, поскольку некоторые операционные системы помещают на такие файлы обязательные блокировки.
- Не полагайтесь на существование в %ENV конкретной переменной окружения и не считайте, что данные в %ENV чувствительны к регистру или сохраняют регистр. Не рассчитывайте на наследование семантики Unix для переменных окружения; в некоторых системах они могут быть видимы для всех других процессов.
- Не используйте сигналы или %SIG.
- Старайтесь избегать поиска имен файлов по шаблону. Вместо этого применяйте opendir, readdir и closedir. (Что касается версии Perl 5.6.0, то базовый поиск имен файлов по шаблону стал значительно более переносимым, но некоторые системы все равно могут приходить в раздражение от

«юниксизмов» интерфейса по умолчанию, если вы попытаетесь соригинальничать.)

Не полагайтесь на конкретные значения номеров ошибок или строк, хранящихся в `$!`.

Межпроцессное взаимодействие (IPC)

Чтобы повысить переносимость, старайтесь не запускать новые процессы. Это означает, что нужно избегать `system`, `exec`, `fork`, `pipe`, `` ` ``, `qx//` и `open` с `|`.

Основную проблему представляют не сами операторы; команды, запускающие внешние процессы, обычно поддерживаются большинством платформ (хотя некоторые не поддерживают ветвление ни в каком виде). Проблемы, скорее всего, могут возникнуть при вызовах внешних программ, для которых семантика имен, адресов, выдачи и аргументов может отличаться от платформы к платформе.

Очень популярно использование кода Perl для открытия канала в *sendmail*, чтобы программы могли отправлять электронную почту:

```
open(MAIL, '|/usr/lib/sendmail -t') or die "cannot fork sendmail: $!";
```

Этот код не будет работать на платформах, где нет *sendmail*. Чтобы создать переносимое решение, используйте для отправки почты один из модулей CPAN, например, `Mail::Mailer` и `Mail::Send` в дистрибутиве MailTools или `Mail::Sendmail`.

Функции Unix System V IPC (`msg*()`, `sem*()`, `shm*()`) не всегда доступны даже на платформах Unix.

Внешние подпрограммы (XS)

Код XS обычно можно заставить выполняться на любой платформе, но библиотеки и файлы заголовков могут оказаться недоступными либо сам код XS может оказаться специфическим для платформы. Если переносимы библиотеки и заголовки, то разумно предположить, что и код XS тоже можно сделать переносимым.

Проблема переносимости другого типа возникает при написании кода XS: наличие компилятора C на платформе конечного пользователя. C привносит свои собственные проблемы переносимости, и написание кода XS сталкивает вас с некоторыми из них. Если писать на чистом Perl, то переносимости добиться легче, потому что процесс конфигурирования Perl изо всех сил старается скрыть от пользователя недостатки переносимости C.[1]

---

[1] Некоторые маргиналы запускают сценарий Perl *Configure* в качестве дешевого развлечения. Известно даже, что устраиваются «Configure races» между конкурирующими системами, на которых разыгрываются крупные суммы. Такая практика в настоящее время признана незаконной в большинстве цивилизованных стран.

Стандартные модули

В целом стандартные модули (модули, увязанные с Perl) работают на всех платформах. Примечательные исключения составляют модули CPAN.pm (в настоящее время он устанавливает соединения с внешними программами, которых может не быть), специфические для платформ модули (например, ExtUtils::MM_VMS) и модули DBM.

Нет модуля DBM, который имелся бы на всех платформах. SDBM_File и прочие обычно доступны во всех версиях для Unix и DOS, но не в MacPerl, где имеются только NBDM_File и DB_File.

Хорошо, что доступным является хотя бы один модуль DBM, и AnyDBM_File будет использовать тот модуль, который сможет найти. При такой неопределенности следует применять только те функции, которые есть во всех реализациях DBM. Например, ограничивайте свои записи размером в 1 Кбайт. Подробности можно найти в документации по модулю AnyDBM_File.

Дата и время

По возможности используйте для представления дат формат ISO-8601 (*"YYYY-MM-DD"*). Строки типа "1987-12-18" легко преобразуются в специфические для системы значения с помощью модуля типа Date::Parse. Список величин типа времени и даты (например, возвращаемый встроенной функцией localtime) может быть преобразован в специфическое для системы представление с помощью Time::Local.

Встроенная функция time всегда возвращает число секунд с начала «эпохи», но операционные системы придерживаются различных мнений относительно ее начала. Во многих системах эпоха началась 1 января 1970 года в 00:00:00 UTC, но на Macintosh она началась на 66 лет раньше, а в VMS – 17 ноября 1858 года в 00:00:00. Поэтому для переносимости времени может потребоваться вычислить смещение начала эпохи:

```
require Time::Local;
$offset = Time::Local::timegm(0, 0, 0, 1, 0, 70);
```

Значением $offset в Unix и Windows всегда будет 0, но на Mac'ах и в VMS это может быть некоторым большим числом. $offset можно затем складывать со значением времени Unix и получать величину, которая в любой системе должна быть одинаковой.

Представлением времени суток и календарной даты в системе можно управлять самыми разными способами. Не рассчитывайте, что временной пояс хранится в $ENV{TZ}. И даже если хранится, не рассчитывайте, что сможете через эту переменную управлять временным поясом.

Многоязычность

Используйте внутри программы Unicode. Включите поддержку преобразования в другие наборы символов и обратно в свои интерфейсы. См. главу 15 «Unicode».

За пределами мира Unicode следует делать как можно меньше допущений о наборах символов и никаких допущений о порядковых значениях символов. Не рассчитывайте, что символы алфавита имеют последовательные порядковые значения. Буквы на нижнем регистре могут идти как до, так и после букв на верхнем регистре; нижний и верхний регистры могут чередоваться, так что `a` и `A` окажутся раньше `b`; акцентированные и прочие национальные символы могут чередоваться случайным образом, так что `д` окажется раньше `b`.

Если ваша программа должна действовать в системе POSIX (довольно широкое предположение), посмотрите дополнительную информацию о национальных настройках POSIX на странице руководства *perllocale*. Национальные установки определяют среди прочего наборы символов и кодировки, а также формат даты и времени. Правильное применение национальных установок сделает вашу программу несколько более переносимой или хотя бы более удобной и дружественной для пользователей, родным языком которых не является английский. Но помните, что национальные установки и Unix пока не очень хорошо сочетаются.

Стиль

Если вам необходимо иметь код, специфический для платформы, постарайтесь поместить его в одном месте, чтобы облегчить перенос на другие платформы. Для определения платформы воспользуйтесь модулем `Config` и специальной переменной `$^O`.

Будьте внимательны с тестами, которые предоставляете вместе со своими модулями и программами. Код модуля может быть полностью переносим, а вот тесты – нет. Это часто происходит, если тесты порождают другие процессы или вызывают внешние программы, которые должны содействовать тестированию, и когда (как было отмечено выше) тесты делают некоторые допущения относительно файловой системы и маршрутов. Следите за тем, чтобы не было зависимости от определенного стиля вывода ошибок, даже при проверке в `$!` «стандартных» ошибок после системного вызова. Лучше использовать модуль `Errno`.

Помните, что хороший стиль – над временем и над культурой, поэтому максимальной переносимости можно добиться, лишь отыскав универсальное в запросах, диктуемых суровой действительностью. Самые рассудительные люди не являются пленниками последней модной причуды; им это не нужно, потому что они, уважая собственную культуру (программистскую или любую другую), не стремятся быть в моде. Мода – это переменная, но стиль – константа.

26

Документация в формате POD

Один из принципов, лежащих в основе архитектуры Perl, гласит, что простые вещи должны быть простыми, а сложные вещи должны быть возможными. Документация должна быть простой.

Perl поддерживает простой формат разметки текста, называемый *pod*, который может использоваться самостоятельно или свободно смешиваться с исходным кодом, образуя встроенную документацию. Pod можно преобразовывать во многие другие форматы для печати или просмотра либо читать непосредственно, потому что он прост.

Pod не так выразителен, как языки вроде XML, LaTeX, *troff* (1) или даже HTML. Это сделано умышленно: мы пожертвовали выразительностью ради простоты и удобства. Некоторые языки разметки таковы, что автору приходится вводить больше разметки, чем самого текста, что излишне осложняет работу и делает почти невозможным чтение. Хороший формат, как хорошая музыка к кинофильму, остается фоном, не отвлекая внимание.

Заставить программистов писать документацию почти так же сложно, как заставить их носить галстук. Pod был задуман таким простым, чтобы даже программист мог (и стал) на нем писать. Мы не утверждаем, что pod достаточно, чтобы написать книгу, хотя для написания этой книги его хватило.

О pod в двух словах

Большинство форматов документов требует, чтобы весь документ был в соответствующем формате. Pod более терпим: можно встраивать pod в файлы

любого типа, полагаясь на *трансляторы pod* для извлечении pod. Некоторые файлы состоят из 100-процентного чистого pod. Но в других файлах, особенно в программах и модулях Perl, фрагменты pod могут быть разбросаны по тем местам, где автор считает это нужным. Perl просто пропускает текст pod при синтаксическом анализе файла перед его выполнением.

Лексический анализатор Perl знает, что нужно начать пропускать текст, когда в том месте, где он рассчитывает найти команду, наталкивается на строку, начинающуюся со знака равенства и идентификатора, например:

```
=head1 Here There Be Pods!
```

Этот, а также последующий текст, вплоть до строки, начинающейся с `=cut`, будет игнорироваться. Это позволяет свободно смешивать исходный код и документацию, например:

```
=item snazzle

The snazzle() function will behave in the most spectacular
form that you can possibly imagine, not even excepting
cybernetic pyrotechnics.

=cut

sub snazzle {
    my $arg = shift;
    ....
}

=item razzle

The razzle() function enables autodidactic epistemology generation.

=cut

sub razzle {
    print "Epistemology generation unimplemented on this platform.\n";
}
```

Другие примеры можно найти в любом стандартном модуле Perl или модуле из CPAN, которые все должны поставляться вместе с pod. И почти во всех модулях pod действительно есть, за исключением тех, где его нет.

Поскольку pod распознается лексическим анализатором Perl и выбрасывается им, вы можете посредством подходящей директивы pod быстро закомментировать фрагмент кода произвольного размера. Используйте блок pod `=for`, чтобы закомментировать один абзац, и пару `=begin/=end` для более крупного участка. О синтаксисе этих директив pod мы расскажем позже. Помните, однако, что в обоих случаях вы остаетесь в режиме pod, пока не вернетесь в компилятор после `=cut`.

```
print "получил 1\n";
```

```
=for commentary
Этот абзац игнорируется всеми, кроме некоего
мифического транслятора "commentary". После его конца вы
все еще в режиме pod, а не в режиме программы.
print "получил 2\n";
=cut

# ok, вот теперь действительно программа
print "получил 3\n";

=begin comment

print "получил 4\n";

все, что здесь,
будет игнорироваться
всеми

print "получил 5\n";

=end comment

=cut

print "получил 6\n";
```

При этом выводится, что получены 1, 3 и 6. Запомните, что эти директивы pod нельзя поместить в произвольное место. Их можно поместить только там, где анализатор рассчитывает обнаружить новую команду, а не просто в середину выражения или куда-либо еще.

С точки зрения Perl вся разметка pod выбрасывается, но с точки зрения трансляторов pod выбрасывается как раз код. Трансляторы pod рассматривают оставшийся текст как последовательность абзацев, разделенных пустыми строками. Все современные трансляторы pod анализируют pod одинаково, используя стандартный модуль `Pod::Parser`. Они отличаются только тем, что именно они выводят, поскольку у каждого транслятора свой особый формат вывода.

Есть три типа абзацев: дословные абзацы (verbatim paragraphs), абзацы команд (command paragraphs) и абзацы прозы (prose paragraphs).

Дословные абзацы

Дословные абзацы используются для буквального текста, который должен выглядеть таким, какой он есть, например, для фрагментов кода. Дословный абзац должен иметь отступ, т. е. начинаться с пробела или символа табуляции. Транслятор должен точно воспроизвести его, обычно моноширинным шрифтом, а символы табуляции предполагаются на границах каждых восьми колонок. Специальных управляющих символов форматирования нет, поэтому нельзя делать выделение курсивом или полужирным шрифтом. Символ < означает литерал < и ничего более.

Директивы pod

Все директивы pod начинаются с символа = и следующего за ним идентификатора. Затем может располагаться произвольный текст любого объема, с которым директива может поступать по своему усмотрению. Единственное требование синтаксиса заключается в том, что весь текст должен быть одним абзацем. В настоящее время распознаются следующие директивы (иногда называемые *командами pod*):

=head1

=head2

...

Директивы =head1, =head2,... создают заголовки заданного уровня. Остальной текст параграфа рассматривается как описание заголовка. Они аналогичны заголовкам разделов и подразделов .SH и .SS в *man*(7) или тегам <H1>...</H1> и <H2>...</H2> в HTML. На самом деле именно в них преобразуются данные директивы трансляторами.

=cut

Директива =cut указывает на конец участка pod. (Далее в документе может быть еще участок pod, но он должен вводиться новой директивой pod.)

=pod

Директива =pod лишь указывает компилятору, что анализ кода должен быть отложен до следующей директивы =cut. Полезна для добавления в документ еще одного абзаца, если текст программы и pod чередуются часто.

=over *NUMBER*

=item *SYMBOL*

=back

Директива =over особым образом начинает раздел, который должен быть списком с элементами, образуемыми директивой =item. Завершается список директивой =back. Число *NUMBER* (если оно указано) указывает форматирующей программе (форматеру), сколько пробелов поместить в отступ. Возможности некоторых форматеров недостаточно богаты, чтобы использовать этот совет, а других, наоборот, слишком богаты, поскольку при работе с пропорциональными шрифтами трудно выровнять все в одну линию, только подсчитывая пробелы. (Обычно считается, что четыре пробела оставляют достаточно места для маркеров или чисел.)

Фактический тип списка задает параметр *SYMBOL* для отдельных элементов. Вот пример списка с маркерами:

```
=over 4

=item *

Mithril armor (кольчуга из мифрила)

=item *
```

```
Elven cloak (эльфийский плащ)

=back
```

А вот нумерованный список:

```
=over 4

=item 1.

First, speak "friend" (скажи "друг").

=item 2.

Second, enter Moria (войди в Морию).

=back
```

А вот именованный список:

```
=over 4

=item armor()

Description of the armor() function

=item chant()

Description of the chant() function

=back
```

Можно создавать вложенные списки одинаковых или разных типов, но при этом нужно придерживаться некоторых основных правил: не используйте `=item` вне блока `=over/=back`; укажите хотя бы одну директиву `=item` в блоке `=over/=back`; и, вероятно, самое важное, сохраняйте единообразие типов элементов внутри данного списка. Либо используйте для каждого элемента `=item *`, чтобы создать маркированный список, либо – `=item 1.`, `=item 2.` и т. д., чтобы создать нумерованный список, либо создайте именованный список посредством `=item foo`, `=item bar` и т. д. Начав с маркеров или чисел, придерживайтесь их, потому что форматерам разрешено использовать первый тип `=item` для определения того, как форматировать список.

Как и во всем, что имеет отношение к pod, результат не может улучшиться от смены транслятора. Одни трансляторы обращают внимание на конкретные числа (или буквы, или римские цифры), следующие за `=item`, другие – нет. Например, текущая версия транслятора *pod2html* весьма бесцеремонна: она вырезает целиком указатели последовательности, даже не пытаясь выяснить тип списка, а потом помещает весь список между тегами `<OL>` и `</OL>`, чтобы броузер мог вывести его как упорядоченный список в HTML. Это не следует считать «функцией» (feature) и, возможно, такое поведение будет со временем исправлено.

=for *TRANSLATOR*

=begin *TRANSLATOR*

=end *TRANSLATOR*

Директивы =for, =begin и =end позволяют включать особые разделы, которые должны передаваться в неизменном виде, но лишь определенным форматерам. Форматеры, которые узнают в *TRANSLATOR* свое имя или псевдоним, обращают внимание на эти директивы; другие полностью их игнорируют. Директива =for указывает, что оставшаяся часть этого абзаца предназначена конкретному транслятору.

```
=for html
<p> This is a <flash>raw</flash> <small>HTML</small> paragraph </p>
```

Парные директивы =begin и =end действуют аналогично =for, но вместо того чтобы принимать только один абзац, они рассматривают весь текст между соответствующими =begin и =end как предназначенный для конкретного транслятора. Вот некоторые примеры:

```
=begin html

<br>Figure 1.<IMG SRC="figure1.png"><br>

=end html

=begin text

  ---------------
  |  foo        |
  |        bar  |
  ---------------

^^^^ Figure 1. ^^^^

=end text
```

Значения *TRANSLATOR*, которые обычно принимаются форматерами, включают в себя roff, man, troff, nroff, tbl, eqn, latex, tex, html и text. Некоторые форматеры воспринимают их частично как синонимы. Ни один транслятор не воспринимает comment – это просто обычное обозначение того, что должно игнорироваться всеми. Таково же предназначение любого нераспознанного слова. Во время написания этой книги авторы часто оставляли для себя пометки с помощью директивы =for later.

Обратите внимание, что =begin и =end могут быть вложенными, но только в том смысле, что самая внешняя пара заставляет рассматривать все в середине как не-pod, даже если там содержатся другие директивы =*word*. Это значит, что когда транслятор видит =begin foo, то он будет либо игнорировать, либо обрабатывать *все* до соответствующей директивы =end foo.

Последовательности pod

Третьим типом абзаца является просто «поток» текста. Это означает, что если абзац начинается не с пробельного символа или знака равенства, то он рассматривается как простой абзац: обычный текст, введенный с возможно меньшим количеством «украшений». Перевод строки рассматривается как эквивалент пробела. Основная работа по приданию абзацу должного внешнего вида возлагается на транслятор, поскольку у программистов есть более важные дела. Предполагается, что трансляторы используют некоторые обычные эвристики – см. раздел «Трансляторы и модули pod» далее в этой главе.

Однако кое-что можно делать явным образом. Внутри обычных абзацев или директив заголовков/элементов списка (но не в дословных абзацах) можно использовать особые последовательности для настройки форматирования. Эти последовательности всегда начинаются с одной заглавной буквы, за которой следует левая угловая скобка, и простираются до парной (не обязательно следующей) правой угловой скобки. В последовательностях могут содержаться другие последовательности.

В pod определены следующие последовательности:

`I<text>`

Текст, выводимый курсивом, предназначен для выделения названий книг, названий кораблей и ссылок на страницы руководства, например, «*perlpod*(1)».

`B<text>`

Текст, выводимый полужирным шрифтом, используется почти исключительно для ключей командной строки и иногда для названий программ.

`C<text>`

Буквальный код, возможно, форматированный шрифтом постоянной ширины типа `Courier`. Не обязателен для простых элементов, которые транслятор должен определить как код, но использовать его следует в любом случае.

`S<text>`

Текст с неразрывными пробелами. Часто окружает другие последовательности.

`L<name>`

Перекрестная ссылка на имя:

`L<name>`

Страница руководства

`L<name/ident>`

Элемент страницы руководства

L<name/"sec">

Раздел в другой странице руководства

L<"sec">

Раздел в данной странице руководства (кавычки не обязательны)

L</"sec">

То же самое

Следующие пять последовательностей те же самые, что и выше, но выводиться будет только *text*, а данные о ссылке скрыты, как в HTML:

L<*text*|*name*>

L<*text*|*name*/*ident*>

L<*text*|*name*/"*sec*">

L<*text*|"*sec*">

L<*text*|/"*sec*">

text не может содержать символы / и | и должен содержать < или > только парами.

F<*pathname*>

Применяется для имен файлов. Обычно выводится так же, как I.

X<entry>

Элемент указателя. Как всегда, что делать, решает транслятор. Спецификация pod это не устанавливает.

E<*escape*>

Именованный символ, аналогично escape-последовательностям HTML:

E<lt>

Литерал < (не обязателен, кроме других внутренних последовательностей и после заглавной буквы)

E<gt>

Литерал > (не обязателен, кроме других внутренних последовательностей)

E<sol>

Литерал / (требуется только в L<>)

E<verbar>

Литерал | (требуется только в L<>)

E<*NNN*>

Символ с номером *NNN*, вероятно, в ISO-8859-1, но может быть в Unicode. Это не должно иметь значения, теоретически...

E<*entity*>

Некая нечисловая сущность HTML, например E<Agrave>.

`Z<>`

Символ нулевой ширины. Удобно помещать впереди последовательностей, которые могут что-то спутать. Например, если в обычной прозе нужно начать строку со знака равенства, можно написать так:

```
Z<>=can you see
```

или в строке с «From», чтобы почтовая программа не поставила впереди >:

```
Z<>From here on out...
```

Чаще всего для указания границ этих последовательностей pod нужен лишь один набор угловых скобок. Однако иногда требуется поместить < или > внутрь последовательности. (Особенно часто это происходит в последовательности `C<>` при задании шрифта постоянной ширины для фрагмента кода.) Как и всегда в Perl, есть несколько способов сделать это. Один способ – просто представить закрывающую скобку `E`-последовательностью:

```
C<$a E<lt>=E<gt> $b>
```

В результате получится `"$a <=> $b"`.

Лучше читается и, возможно, более прост способ, при котором используется альтернативный набор ограничителей, который не требует escape-последовательностей для угловых скобок. Применение двойных угловых скобок (`C<<` *`stuff`* `>>`) допускается при условии, что непосредственно за открывающим ограничителем и непосредственно перед закрывающим ограничителем имеется пробельный символ. Например, можно написать так:

```
C<< $a <=> $b >>
```

Количество повторяющихся угловых скобок может быть любым, лишь бы они были парными, а пробельный символ непосредственно следовал за последней < слева и непосредственно перед первой > справа. Поэтому можно писать и так:

```
C<<< $a <=> $b >>>
C<<<< $a <=> $b >>>>
```

В любом случае будет выводиться `$a <=> $b` шрифтом постоянной ширины.

Лишние пробелы внутри по обоим концам удаляются, поэтому если вам нужны пробелы, поставьте их снаружи. Кроме того, два внутренних участка дополнительных пробелов не перекрывают друг друга, поэтому если в начале заключенного в кавычки текста стоят двойные угловые скобки >>, они не воспринимаются как закрывающий разделитель:

```
The C<< >> >> right shift operator.
```

Это порождает текст «The >> right shift operator.».

Обратите внимание, что pod-последовательности могут быть вложенными. Это значит, что можно написать `"The I<Santa MarE<iacute>a> left port already"`, чтобы получить «The *Santa María* left port already», или `"B<touch> S<B<-t> I<time>> I<file>"`, чтобы получить «***touch –t time file***», и это будет работать правильно.

Трансляторы и модули Pod

Perl поставляется с несколькими трансляторами pod, которые преобразуют документы pod (или встроенный pod в документах других типов) в различные форматы. Все должны сбрасывать 8-й бит.

pod2text

Преобразует pod в текст. Обычно на выходе получается 7-разрядный текст ASCII; 8-разрядный, если он был 8-разрядным на входе, а если используются последовательности вида `LE<uacute>thien` (для *Lúthien*) или `EE<auml>rendil` (для *Eärendil*), то текст на выходе имеет кодировку ISO-8859-1 (или Unicode).

Если у вас есть файл, содержащий pod, то самый простым (но, возможно, не самым красивым) способом просмотреть только форматированный pod будет следующий:

```
% pod2text File.pm | more
```

Опять же, pod должен быть читаемым без форматирования.

pod2man

Преобразует pod в формат страницы руководства Unix, пригодный для просмотра с помощью *nroff*(1) или создания печатных экземпляров с помощью *troff*(1). Например:

```
% pod2man File.pm | nroff -man | more
```

или

```
% pod2man File.pm | troff -man -Tps -t > tmppage.ps
% ghostview tmppage.ps
```

и для печати:

```
% lpr -Ppostscript tmppage.ps
```

pod2html

Преобразует pod в HTML, который можно просматривать в любимом броузере:

```
% pod2html File.pm > tmppage.html
% lynx tmppage.html
% netscape -remote "openURL(file:`pwd`/tmppage.html)"
```

Последнее представляет собой прием для *netscape*, который действует, если *netscape* уже где-то выполняется, и указывает ему на необходимость загрузки страницы. В противном случае просто вызовите его так же, как *lynx*.

pod2latex

Преобразует pod в LaTeX.

Для других форматов можно найти в CPAN другие трансляторы.

Режим работы трансляторов по умолчанию зависит от выходного формата. Например, если у вас есть в pod такой абзац прозы:

```
This is a $variable right here
```

то *pod2html* превратит его в:

```
This is a <STRONG>$variable</STRONG> right here
```

но *pod2text* ничего украшать не будет, потому что знака доллара должно быть достаточно для возможности чтения.

Составляя pod, следует держаться поближе к простому тексту, не злоупотребляя явными разметками. Решать, как следует представлять текст, должен каждый отдельный транслятор. Это означает, что транслятор должен определять, как создавать парные кавычки, как вводить и выравнивать текст, как найти более мелкий шрифт для слов, целиком написанных заглавными буквами, и т. д. Поскольку трансляторы написаны для обработки документов Perl, в большинстве своем[1] они должны также распознавать следующие элементы и выводить их надлежащим образом:

- `FILEHANDLE`
- `$scalar`
- `@array`
- `function()`
- `manpage(3r)`
- `somebody@someplace.com`
- `http://foo.com/`

С Perl поставляются также несколько стандартных модулей для анализа и преобразования pod, в том числе `Pod::Checker` (и соответствующая утилита *podchecker*) для проверки синтаксиса документов pod, `Pod::Find` для поиска документов pod в деревьях каталогов и `Pod::Parser` для создания собственных утилит работы с pod.

Обратите внимание, что трансляторы pod должны рассматривать только абзацы, начинающиеся с директивы pod (это облегчает анализ), в то время как

[1] Если разрабатывается транслятор pod общего назначения, а не только для кода Perl, то критерии могут быть иными.

компилятор фактически умеет искать escape-последовательности pod даже в середине абзаца. Это значит, что следующий секретный фрагмент secret stuff будет игнорироваться и компилятором, и трансляторами:

```
$a=3;
=secret stuff
warn "Neither POD nor CODE!?"
=cut back
print "got $a\n";
```

Не надейтесь, что эта команда warn всегда будет попадать в pod. Не все трансляторы pod хорошо ведут себя в этом отношении, да и компилятор в один прекрасный день может стать разборчивее.

Создание собственных инструментов для работы с pod

Pod был разработан так, чтобы на нем, прежде всего, было легко писать. В качестве дополнительного преимущества простота pod дает возможность создавать простые средства для его обработки. Для того чтобы найти директивы pod, установите разделитель записей в режим абзацев (вероятно, с помощью ключа -00) и обращайте внимание только на те абзацы, которые похожи на pod.

Вот, например, простая программа *olpod*, воспроизводящая структуру pod:

```
#!/usr/bin/perl -l00n
# olpod - outline pod
next unless /^=head/;
s/^=head(\d)\s+/ ' ' x ($1 * 4 - 4)/e;
print $_, "\n";
```

Если выполнить ее с данной главой, то можно получить нечто вроде:

```
Документация в формате pod
    О pod в двух словах
        Дословные абзацы
        Директивы pod
        Последовательности pod
    Трансляторы и модули pod
    Создание собственных инструментов для работы с pod
    Ловушки pod
    Документирование программ на Perl
```

Приведенная программа создания структуры pod на самом деле не обращала внимания на то, были ли это действительные блоки pod или нет. Поскольку pod и не-pod могут быть перемешаны внутри одного файла, то запуск средств общего назначения для поиска или анализа файла целиком не всегда имеет смысл. Но это и не имеет значения, с учетом того, как просто пишутся сред-

ства для работы с pod. Вот средство, которое учитывает разницу между pod и не-pod и выводит только pod:

```
#!/usr/bin/perl -00
# catpod - cat только pod
while (<>) {
    if (! $inpod) { $inpod = /^=/;             }
    if ($inpod)   { $inpod = !/^=cut/; print; }
} continue {
    if (eof)      {  close ARGV; $inpod = ''; }
}
```

Можно использовать эту программу с другой программой или модулем Perl, а затем перенаправить вывод по конвейеру в другой инструмент. Например, если имеется программа *wc*(1)[1] для подсчета строк, слов и символов, можно перенаправить вывод с *catpod*, чтобы использовать при подсчете только pod:

```
% catpod MyModule.pm | wc
```

Есть масса ситуаций, в которых pod позволяет писать примитивные инструменты, тривиально используя простой бесхитростный Perl. Теперь у нас в качестве компонента есть *catpod*, и можно создать еще одно средство, которое показывает код с отступами:

```
#!/usr/bin/perl -n00
# podlit - вывести литеральные блоки с отступом из входных данных pod
print if /^\s/;
```

Как это можно использовать? Например, можно реализовать проверки *perl -wc* для кода, находящегося в документе. Или создать разновидность *grep*(1)[2], которая рассматривает только примеры кода:

```
% catpod MyModule.pm | podlit | grep funcname
```

Такая философия инструментов и фильтров, состоящих из взаимозаменяемых (и допускающих независимое тестирование) частей, представляет собой чрезвычайно простой и мощный подход к разработке повторно используемых программных компонентов. Это разновидность лени, состоящая в том, чтобы находить минимальные решения, позволяющие решать задачи дня – по крайней мере, некоторые.

В других случаях, однако, это может оказаться контрпродуктивным. Иногда написание инструмента с нуля требует больше труда, иногда меньше. Для того, что было нами показано выше, собственное мастерство Perl в обработке текста делает целесообразным применение грубой силы. Но не всегда так бывает. Занимаясь pod, вы могли заметить, что его директивы просто анализировать, но его последовательности иногда оказываются несколько риско-

---

[1] Ее можно найти в Perl Power Tools из каталога CPAN *scripts*.

[2] А если у вас нет *grep*, см. предыдущую сноску.

ванными. Последовательности могут быть вложены в другие последовательности и использовать ограничители переменной длины, хотя некоторые «полукорректные» трансляторы этого не поддерживают.

Избавляя от необходимости писать самостоятельно весь этот код для синтаксического разбора, лень подсказывает другое решение. Для этого годится стандартный модуль `Pod::Parser`. Особенно полезен он для сложных задач, в которых, например, требуется действительно анализировать внутренние участки абзацев, преобразовывать в альтернативные выходные форматы и т. д. В сложных случаях с этим модулем легче работать, поскольку в итоге приходится писать меньше кода. Удобство и в том, что хитрый синтаксический анализ оказывается уже проделанным. На самом деле это тот же принцип, что и при использовании *catpod* в конвейере.

Подход модуля `Pod::Parser` к своей задаче интересен. Это объектно-ориентированный модуль иного типа, чем большинство тех, которые вы видели в этой книге. Его основная задача не столько в том, чтобы предоставлять объекты для непосредственного использования, сколько в том, чтобы предоставить базовый класс, на основе которого можно строить другие классы.

Мы создаем собственный класс и осуществляем наследование от `Pod::Parser`. Затем объявляем подпрограммы, служащие методами обратного вызова, которые будут вызываться анализатором из нашего родительского класса. Такой способ программирования весьма отличается от процедурных программ, приводившихся нами ранее. В некотором смысле это ближе к декларативному стилю программирования, т. к. решение задачи состоит в том, что мы регистрируем функции и разрешаем другим объектам их вызывать. Сложный алгоритм программы выполняется где-то в другом месте, мы просто предоставляем нечто типа «plug-and-play».

Вот переработанная версия рассмотренной выше программы *catpod*, в которой на этот раз используется модуль `Pod::Parser` для создания собственного подкласса:

```
#!/usr/bin/perl
# catpod2, класс и программа

package catpod_parser;
use Pod::Parser;
@ISA = qw(Pod::Parser);
sub command {
    my ($parser, $command, $paragraph, $line_num) = @_;
    my $out_fh = $parser->output_handle();
    $paragraph .= "\n" unless substr($paragraph, -1) eq "\n";
    $paragraph .= "\n" unless substr($paragraph, -2) eq "\n\n";
    print $out_fh "=$command $paragraph";
}

sub verbatim {
    my ($parser, $paragraph, $line_num) = @_;
    my $out_fh = $parser->output_handle();
```

```
    print $out_fh $paragraph;
}

sub textblock {
    my ($parser, $paragraph, $line_num) = @_;
    my $out_fh = $parser->output_handle();
    print $out_fh $paragraph;
}

sub interior_sequence {
    my ($parser, $seq_command, $seq_argument) = @_;
    return "$seq_command<$seq_argument>";
}

if (!caller) {
    package main;
    my $parser = catpod_parser::->new();
    unshift @ARGV, '-' unless @ARGV;
    for (@ARGV) { $parser->parse_from_file($_); }
}
1;
__END__

=head1 NAME
здесь документация, описывающая новую программу catpod
```

Как видите, эта версия стала длиннее и сложнее. Кроме того, она легче расширяема, поскольку все, что мы должны сделать для того, чтобы наш подкласс действовал иначе, чем его базовый класс, – это подключить свои собственные методы.

Фрагмент в конце, содержащий `!caller`, проверяет, как используется файл: как модуль или как программа. Если как программа, то `caller` нет. Тогда запускается собственный анализатор (с помощью унаследованного метода `new`), которому передаются аргументы командной строки. Если имена файлов не передавались, используется стандартное устройство ввода, как и в предыдущей версии.

Код модуля завершают маркер `__END__`, пустая строка без пробельных символов и собственная pod-документация программы/модуля. Это пример одного файла, представляющего собой *и* программу, *и* модуль, *и* собственную документацию. Возможно, он является также чем-нибудь еще.

Ловушки pod

Pod весьма прост, но все же в некоторых местах можно допустить промахи:

- Довольно легко пропустить закрывающую угловую скобку.
- Довольно легко пропустить замыкающую директиву `=back`.

- Нетрудно случайно вставить пустую строку в середине длинной директивы `=for comment`. Попробуйте использовать вместо нее `=begin/=end`.
- При ошибке во вводе одного из тегов в паре `=begin/=end` будет съедена вся оставшаяся часть вашего файла (как pod). Попробуйте вместо этого применить директиву `=for`.
- Трансляторы pod требуют, чтобы абзацы разделялись абсолютно пустыми строками, т. е. двумя или более последовательными символами перевода строки (`\n`). Если в строке есть пробелы или символы табуляции, она не считается пустой. В результате два или более абзаца рассматриваются как один.
- Понятие «ссылки» в pod не определено, и каждый транслятор сам решает, что с ней делать. (Тот, кто начинает подозревать, что большинство решений pod предоставляет делать трансляторам, прав.) Трансляторы часто добавляют слова вокруг ссылки `L<>`, поэтому `"L<foo(1)>"` может, например, стать «the *foo*(1) manpage». Поэтому не следует писать вещи типа `"the L<foo> manpage`, если требуется, чтобы оттранслированный документ осмысленно читался: в итоге получится «the the *foo*(1) manpage manpage».

  Если необходим полный контроль над текстом, выступающим в качестве ссылки, используйте формат `L<show this text|foo>`.

Стандартная программа *podchecker* проверяет синтаксис pod и выводит ошибки или предупреждения. Например, она обнаруживает неизвестные последовательности pod и якобы пустые строки, содержащие пробельные символы. Все же рекомендуется пропустить свой документ через два или более разных транслятора pod и проверить результаты. Некоторые обнаруженные проблемы могут просто быть своеобразными особенностями конкретных трансляторов, которые просто не всем нравятся.

И, как всегда, Все Может Измениться по Прихоти Случайного Хакера (Everything is Subject To Change at the Whim of the Random Hacker).

Документирование программ Perl

Мы надеемся, что читатель документирует свой код независимо от того, является ли он Случайным Хакером. Если да, то желательным может оказаться включение в pod таких разделов:

`=head1 NAME`

Имя программы или модуля.

`=head1 SYNOPSIS`

Одна строка, описывающая то, что делает (предположительно) программа или модуль.

`=head1 DESCRIPTION`

Массив документации. («Массив» подходит в данном контексте.)

=head1 AUTHOR

Кто вы такой. (Или псевдоним, если вам стыдно за свою программу.)

=head1 BUGS

Что вы делали неправильно (и почему на самом деле вы в этом не виноваты).

=head1 SEE ALSO

Где можно найти дополнительную информацию (чтобы справиться с ошибками в программе).

=head1 COPYRIGHT

Уведомление об авторских правах. Если вы хотите заявить об авторских правах, можно сказать что-то вроде:

```
Copyright 2013, Randy Waterhouse. All Rights Reserved.
```

Во многих модулях также добавляется:

```
This program is free software. You may copy or
redistribute it under the same terms as Perl itself.
[Эта программа распространяется свободно. Вы можете копировать
или распространять ее на тех же условиях, что и сам Perl]
```

Одно предостережение: помещая свой pod в конце файла и используя лексемы __END__ или __DATA__, не забудьте поместить пустую строку перед первой директивой pod:

```
__END__

=head1 NAME

Modern - I am the very model of a modern major module
```

Если перед =head1 отсутствует пустая строка, трансляторы pod не обнаруживают вашу (пространную, точную и грамотную) документацию.

27

Культура Perl

Эта книга является частью культуры Perl, и мы не можем надеяться, что в нее удастся поместить все, что мы знаем о культуре Perl. Немного истории и немного искусства – некоторые сказали бы «очень немного искусства» – позволят нам раздразнить ваш аппетит. Значительно больше о культуре Perl сказано на *www.perl.org* и *www.perl.com*. (Ларри хранит копии всего своего (официального) бреда на *www.wall.org/~larry*.) Или просто познакомьтесь с какими-нибудь другими программистами Perl. Мы не можем сказать, какого сорта люди вам попадутся: едва ли не единственная общая черта характера программистов Perl – это патологическая готовность прийти на помощь.

История развития

Чтобы понять, почему Perl такой, какой он есть (или каким он не является), надо сначала разобраться, откуда вообще взялся Perl. Поэтому достанем старую, запылившуюся книгу истории...

Когда-то давным-давно, в 1986 году, Ларри был системным программистом в проекте по разработке глобальных сетей с многоуровневой защитой. Он отвечал за структуру, состоявшую из трех VAX и трех Sun на западном побережье, соединенных на скорости 1200 бод через последовательную линию с шифрованием с аналогичной конфигурацией на восточном побережье. Поскольку основной задачей Ларри была поддержка (в проекте он участвовал не как программист, а просто как системный гуру), то он смог воспользоваться тремя своими добродетелями (ленью, нетерпением и высокомерием),

чтобы разработать и усовершенствовать всевозможные полезные инструменты, например *rn*, *patch* и *warp*.[1]

Однажды, как раз когда Ларри разорвал *rn* в клочья и разбросал их по своему каталогу, к нему вошел Большой Менеджер и сказал: «Ларри, нам нужна система управления конфигурацией и контроля для всех шести VAX и всех шести Sun. Через месяц. Займись-ка этим!»

Итак, Ларри, который никогда не уклонялся от работы, задал себе вопрос: как лучше всего получить систему управления компьютерами на восточном и западном побережьях, позволяющую видеть отчеты о проблемах, возникших в обоих местах, давать разрешения и осуществлять контроль, и при этом не писать все с самого начала. Ответ пришел в виде одного слова: B-news.[2]

Ларри взялся за дело и установил на этих машинах программы работы с телеконференциями, добавив две команды управления: «append», чтобы выполнять добавление к существующей статье, и «synchronize», чтобы номера статей были одинаковыми на обоих побережьях. Управление компьютерами должно было осуществляться с помощью систем управления версиями (Revision Control System, RCS), а одобрения и представления должны были осуществляться с помощью новостей и *rn*. Пока все прекрасно.

Затем Большой Менеджер попросил представить ему отчеты. Сообщения хранились в отдельных файлах на главной машине, а файлы были связаны массой перекрестных ссылок. Первой мыслью Ларри было: надо использовать *awk*. К несчастью, в то время *awk* не мог справляться с открытием и закрытием многих файлов на основе информации, содержащейся в этих файлах. Ларри не захотел писать инструментальное средство для решения специальной задачи. В результате появился новый язык.

Этот новый инструмент изначально не назывался Perl. Ларри обсуждал какие-то имена с коллегами и помощниками (Дэном Фэйджином (Dan Faigin), автором этого рассказа, и Марком Биггаром (Mark Biggar), своим родственником, который также очень помог ему в начальной разработке). Ларри рассмотрел и отверг все трех- и четырехбуквенные слова из словаря. Рассматривалось также имя «Gloria», в честь его возлюбленной (и жены). Но затем он решил, что это вызвало бы ненужные недоразумения в семье.

---

[1] Примерно в это время Ларри понял фразу «feeping creaturism» (В «Новом словаре хакера» Эрика С. Реймонда чудесно сказано, что, во-первых, этот термин происходит от «creeping featurism», а во-вторых, «...четкого определения ... нет, но любой хакер поймет, о чем идет речь». – *Примеч. перев.*) в отчаянной попытке оправдать на основании биологической необходимости свое непреодолимое стремление «добавить еще одну функцию». В конце концов, если жизнь просто слишком сложна, не относится ли это и к программам? Особенно к таким программам, как *rn*, которые следовало бы рассматривать как передовые проекты искусственного интеллекта, поскольку они могут вместо вас читать сообщения телеконференций. Правда, некоторые считают, что и программа *patch* уже *слишком* интеллектуальна.

[2] То есть вторую реализацию транспортного программного обеспечения Usenet.

Затем возникло название «Pearl», превратившееся в современное «Perl», отчасти потому, что Ларри обнаружил ссылку на другой язык с названием PEARL, но главным образом потому, что он слишком ленив, чтобы каждый раз вводить пять букв. И конечно, потому, что Perl подходил на роль слова из четырех букв. (Однако следы прежнего написания можно обнаружить в толковании акронима: «Practical Extraction *And* Report Language».)

В этом раннем Perl отсутствовали многие особенности сегодняшнего Perl. Поиск по шаблону и указатели файлов были, скаляры и форматы тоже, но было очень мало функций, довольно увечная реализация регулярных выражений, заимствованная из *rn*, и не было ассоциативных массивов. Руководство имело объем всего в 15 страниц. Но Perl был быстрее, чем *sed* и *awk*, и стал использоваться в других приложениях проекта.

Но Ларри был нужен везде. В один прекрасный день пришел еще один Большой Менеджер, который сказал: «Ларри, займись поддержкой исследовательских работ». «O'kay», – сказал Ларри. Он прихватил с собой Perl и обнаружил, что тот стал хорошим средством системного администрирования. Он воспользовался прекрасным пакетом регулярных выражений Генри Спенсера (Henry Spencer) и перекромсал его так, что Генри старался не думать об этом во время еды. Затем Ларри добавил некоторые вещи, которые были нужны ему, и некоторые вещи, которые были нужны другим. Он разместил Perl в сети.[1] Остальное, как говорят, уже история.[2]

А развивается она примерно так. Perl 1.0 был выпущен 18 декабря 1987 года; некоторые относятся к дню рождения Perl серьезно. Perl 2.0 следует в июне 1988, и Рэндал Шварц (Randal Schwartz) создает свою легендарную подпись «Just Another Perl Hacker». В 1989 году Том Кристиансен (Tom Christiansen) представляет первый общедоступный учебник Perl на Usenix в Балтиморе. В версии Perl 3.0 в октябре 1989 язык впервые выпускается и распространяется на условиях GNU Public License.

В марте 1990 Ларри пишет первое стихотворение – Perl Poem (см. следующий раздел). Затем он и Рэндал пишут первое издание этой книги – The Pink Camel – «розового верблюда»; она была опубликована в начале 1991 года. Одновременно происходит выпуск Perl 4.0; вместе с GPL он содержит Артистическую Лицензию (Artistic License).

Представление давно ожидавшегося Perl 5 происходит в октябре 1994. Это полностью переписанный Perl, в который вошли объекты и модули. Пришествие Perl 5 даже удостоилось освещения в «The Economist». В 1995 сооб-

---

[1] Более поразительно, что он продолжал делать новые версии, когда начал работать в Jet Propulsion Lab, затем в NetLabs и Seagate. В настоящее время большую часть действительной работы выполняют другие, а Ларри делает вид, что работает в O'Reilly & Associates (небольшой компании, издающей памфлеты о компьютерах и подобной дребедени).

[2] Историческая справка. Когда началась работа над Perl, *rn* как раз была разобрана «по кирпичику» для капитального ремонта. Начав работать над Perl, Ларри не прикоснулся к *rn*. Она по-прежнему разобрана на кусочки. Время от времени Ларри грозится переписать *rn* на Perl, но это несерьезно.

обществу Perl официально представлен CPAN. Джон Орвант (Jon Orwant) начинает выпускать *The Perl Journal* в 1996 году. Осенью того же года, после длительного созревания, вышло второе издание этой книги – The Blue Camel – «голубой верблюд». Первая Конференция O'Reilly по Perl (TPC) проводится в Сан Хосе, штат Калифорния, летом 1997. Значительные события теперь происходят почти ежедневно, поэтому продолжение истории см. в Perl Timeline на CPAST – Comprehensive Perl Arcana Society Tapestry (*history.perl.org*).

Поэзия Perl

Поэтическая выдумка, приведенная в рамочке, появилась в Usenet в День смеха 1 апреля 1990 года. Она приводится здесь без комментариев, просто чтобы показать, как отвратительны в действительности метафоры типичного языка программирования. Как и всего, что кажется имеющим литературную ценность. Ларри особенно удовлетворен тем, что «Black Perl», написанный изначально для Perl 3, больше не пропускается синтаксическим анализатором в Perl 5.

Собрание сочинений, приведенное Ларри, к счастью, оставлено в тени сочинениями королевы поэзии Perl Шэрон Хопкинс (Sharon Hopkins). Она написала много стихотворений Perl, а также статью о поэзии Perl, которую представила на технической конференции Usenix зимой 1992 года, озаглавленную «Camels and Needles: Computer Poetry Meets the Perl Programming Language» (Верблюды и иголки: встреча компьютерной поэзии и языка программирования Perl). (Статью можно найти в *misc/poetry.ps* в CPAN.) Шэрон не только самая плодовитая из поэтов Perl, но и самая публикуемая; следующее ее стихотворение было опубликовано в *Economist* и *Guardian*:

```
#!/usr/bin/perl
APPEAL:
listen (please, please);
open yourself, wide; join (you, me),
connect (us,together),
tell me.
do something if distressed;
    @dawn, dance;
    @evening, sing;
    read (books,$poems,stories) until peaceful;
    study if able; write me if-you-please;
sort your feelings, reset goals, seek (friends, family, anyone);
        do*not*die (like this) if sin abounds;
keys (hidden), open (locks, doors), tell secrets;
do not, I-beg-you, close them, yet.
                            accept (yourself, changes), bind (grief, despair);
require truth, goodness if-you-will, each moment;
select (always), length(of-days)
# listen (a perl poem)
# Sharon Hopkins
# rev. June 19, 1995
```

Perl Poetry

Article 970 of comp.lang.perl:
Path: jpl-devvax!pl-dexxav!lwall
From: lwall@jpl-dexxav.JPL.NASA.GOV (Larry Wall)
Newsgroups: news.groups,rec.arts.poems,comp.lang.perl
Subject: CALL FOR DISCUSSION: comp.lang.perl.poems
Message-ID: 0401@jpl-devvax.JPL.NASA.GOV
Date: 1 Apr 90 00:00:00 GMT
Reply-To: lwall@jpl-devvax.JPL.NSAS.GOV (Larry Wall)
Organization: Jet Prepulsion Laboratory, Pasadena, CA
Lines: 61

До моего сведения дошло, что есть отчаянная потребность в таком месте, где люди могли бы одновременно выразить эмоциональную и техническую стороны своей натуры. Несколько человек прислали мне вещи, которые не вписываются ни в одну из имеющихся телеконференций. Возможно, это потому, что я недавно писал в comp.lang.perl и rec.arts.poems, но люди пишут стихотворения на Perl и спрашивают меня, куда их нужно отправлять. Вот образцы:
От студента, сдающего выпускные экзамены, такое хокку:

```
study, write, study,
do review (each word) if time.
close book. sleep? what's that?
```

А вот от человека из Fort Lauderdale:

```
sleep, close together,
sort of sin each spring & wait;
50% die
```

Некто, пожелавший остаться неизвестным, написал такой образец «черного Перла». (Автор стихотворения «Pearl» был бы, несомненно, в шоке.)

```
BEFOREHAND: close door, each window & exit;  wait until time.
    open spellbook, study, read (scan, select, tell us);
write it, print the hex while each watches,
    reverse its length, write again;
    kill spiders, pop them, chop, split, kill them.
       unlink arms, shift, wait & listen (listening, wait),
sort the flock (then, warn the "goats" & kill the "sheep");
    kill them, dump qualms, shift moralities,
    values aside, each one;
        die sheep! die to reverse the system
        you accept (reject, respect);
next step,
    kill the next sacrifice, each sacrifice,
    wait, redo ritual until "all the spirits are pleased";
    do it ("as they say").
do it(*everyone***must***participate***in***forbidden**s*e*x*).
return last victim; package body;
    exit crypt (time, times & "half a time") & close it,
    select (quickly) & warn your next victim;
AFTERWORDS: tell nobody.
    wait, wait until time;
    wait until next year, next decade;
        sleep, sleep, die yourself,
        die at last
```

Я попробовал, и это действительно проходит синтаксический анализ в Perl. Вроде бы ничего полезного оно не делает, чем я доволен... Я предлагаю создать конференцию comp.lang.perl.poems в качестве места для таких произведений, чтобы не забивать конференции perl или poems тем, что может быть неинтересно ни одной из них. Либо мы должны создать rec.arts.poems.perl для таких сочинений, как приведенные, которые просто проходят синтаксический анализ и ничего полезного не делают. (В конце концов, в rec.arts.poems существует прецедент.) А затем создать comp.lang.perl.poems для стихотворений, которые на самом деле что-то выполняют, как следующее хокку моего собственного сочинения:

```
print STDOUT q
Just another Perl hacker,
unless $spring
```

Larry Wall lwall@jpl-devvax.jpl.nasa.gov

V

Справочный материал

28

Специальные имена

Эта глава посвящена переменным, которые имеют в Perl особые значения. Для большинства имен, содержащих знаки пунктуации, есть хорошие мнемоники или аналоги в какой-нибудь оболочке (или и то и другое). Но если вы хотите использовать в качестве синонимов длинные имена переменных, просто скажите в начале программы:

```
use English;
```

В результате в текущем пакете для всех коротких имен будут установлены псевдонимы в виде длинных имен. У некоторых из этих переменных даже есть средние имена, обычно заимствованные из *awk*. Большинство, в конечном счете, останавливает выбор на коротких именах, по крайней мере, для самых употребительных переменных. В данной книге мы систематически используем короткие имена, но часто указываем и длинные (в скобках), чтобы их легко можно было найти в данной главе.

Семантика этих переменных может быть совершенно магической. (Как создавать собственную магию, читайте в главе 14 «Связанные переменные».) Некоторые из этих переменных доступны только для чтения. При попытке присвоить им значения возникает исключительная ситуация.

Ниже мы сначала приводим полный список переменных и функций, которым присвоено в Perl особое значение, сгруппировав их по типу, чтобы можно было находить переменные, в точном названии которых вы не уверены. Затем мы рассказываем о каждой переменной в алфавитном порядке их правильных имен (или наименее неправильных).

Специальные имена, сгруппированные по типам

Мы вольно употребили слово «тип» – представленные разделы группируют переменные скорее по области видимости.

Специальные переменные регулярных выражений

Следующие специальные переменные, связанные с поиском по шаблону, видны во всей динамической области видимости, в которой происходит поиск по шаблону (за исключением устаревшей `$*`). Иными словами, ведут себя так, как будто они объявлены посредством `local`, и вам не нужно самим объявлять их таким образом. См. главу 5 «Поиск по шаблону».

```
$*
$digits
@+ (@LAST_MATCH_END)
@- (@LAST_MATCH_START)
$+ ($LAST_PAREN_MATCH)
$^R ($LAST_REGEXP_CODE_RESULT)
$& ($MATCH)
$' ($POSTMATCH)
$` ($PREMATCH)
```

Переменные указателя файла

Эти специальные переменные не требуется определять как `local`, потому что они всегда указывают на значение, относящееся к выбранному в данный момент указателю файла вывода, – каждый указатель файла сохраняет свой собственный набор значений. Когда вы выбираете новый указатель файла, то прежний запоминает значения, которые имели эти переменные, а переменные начинают отражать значения нового указателя файла. См. также описание модуля `FileHandle` в главе 32 «Стандартные модули».

```
$| ($AUTOFLUSH, $OUTPUT_AUTOFLUSH)
$- ($FORMAT_LINES_LEFT)
$= ($FORMAT_LINES_PER_PAGE)
$~ ($FORMAT_NAME)
$% ($FORMAT_PAGE_NUMBER)
$^ ($FORMAT_TOP_NAME)
```

Специальные переменные пакетов

Эти переменные существуют отдельно в каждом пакете. Нужды в их локализации не должно быть, поскольку `sort` автоматически делает это с `$a` и `$b`, а остальные, скорее всего, лучше оставить в покое (хотя потребуется объявить их, если вы используете директиву `strict`).

```
$a
$b
@EXPORT
@EXPORT_OK
%EXPORT_TAGS
%FIELDS
@ISA
$VERSION
```

Специальные переменные для всей программы

Эти переменные действительно являются глобальными в самом широком смысле – они означают одну и ту же вещь в каждом пакете, т. к. все перенаправляются в пакет `main`, если не заданы как квалифицированное имя (за исключением `@F`, которая является специальной в `main`, но принудительно к ней не отсылаются). Если вам требуется временный экземпляр одной из них, локализуйте его в текущей динамической области видимости.

```
%ENV
%INC
%SIG
%!
%^H

@_
@ARGV
@F
@INC

$_ ($ARG)
$0 ($PROGRAM_NAME)
$ARGV

$! ($ERRNO, $OS_ERROR)
$" ($LIST_SEPARATOR)
$#
$$ ($PID, $PROCESS_ID)
$( ($GID, $REAL_GROUP_ID)
$) ($EGID, $EFFECTIVE_GROUP_ID)
$, ($OFS, $OUTPUT_FIELD_SEPARATOR)
$. ($NR, $INPUT_LINE_NUMBER)
$/ ($RS, $INPUT_RECORD_SEPARATOR)
$: ($FORMAT_LINE_BREAK_CHARACTERS)
$; ($SUBSEP, $SUBSCRIPT_SEPARATOR)
$< ($UID, $REAL_USER_ID)
$> ($EUID, $EFFECTIVE_USER_ID)
$? ($CHILD_ERROR)
$@ ($EVAL_ERROR)
$[
$\ ($ORS, $OUTPUT_RECORD_SEPARATOR)
$]  ($OLD_PERL_VERSION)
$^A ($ACCUMULATOR)
$^C ($COMPILING)
$^D ($DEBUGGING)
$^E ($EXTENDED_OS_ERROR)
$^F ($SYSTEM_FD_MAX)
$^H
$^I ($INPLACE_EDIT)
$^L ($FORMAT_FORMFEED)
$^M
$^O ($OSNAME)
$^P ($PERLDB)
$^R ($LAST_REGEXP_CODE_RESULT)
$^S (EXCEPTIONS_BEING_CAUGHT)
$^T ($BASETIME)
$^V ($PERL_VERSION)
$^W($WARNING)
${^WARNING_BITS}
${^WIDE_SYSTEM_CALLS}
$^X ($EXECUTABLE_NAME)
```

Специальные указатели файлов для пакетов

За исключением указателя DATA, который всегда относится к пакету, следующие указатели файлов относятся к main, если не квалифицированы полностью именем другого пакета:

```
_ (подчеркивание)
ARGV
ARGVOUT
DATA
STDIN
STDOUT
STDERR
```

Специальные функции для пакетов

Следующие имена подпрограмм имеют для Perl особое значение. Они всегда вызываются неявно при возникновении некоторых событий, таких как доступ к связанным переменным или попытка вызвать неопределенную функцию. Мы не описываем их в этой главе, поскольку они интенсивно освещаются на протяжении всей книги.

Перехватчик вызова неопределенной функции (см. главу 10 «Пакеты»):

```
AUTOLOAD
```

Завершение отживших свое объектов (см. главу 12 «Объекты»):

```
DESTROY
```

Объекты исключительных ситуаций (см. die в следующей главе):

```
PROPAGATE
```

Функции автоматической инициализации и автоматической уборки (см. главу 18 «Компиляция»):

```
BEGIN, CHECK, INIT, END
```

Методы связывания (см. главу 14 «Связанные переменные»):

```
BINMODE, CLEAR, CLOSE, DELETE, EOF, EXISTS, EXTEND, FETCH, FETCHSIZE, FILENO, FIRSTKEY,
GETC, NEXTKEY, OPEN, POP, PRINT, PRINTF, PUSH, READ, READLINE, SEEK, SHIFT, SPLICE,
STORE, STORESIZE, TELL, TIEARRAY, TIEHANDLE, TIEHASH, TIESCALAR, UNSHIFT, WRITE
```

Специальные переменные в алфавитном порядке

Мы привели эти переменные в алфавитном порядке их длинных имен. Если вы не знаете длинного имени переменной, можете найти его в предыдущем разделе. (Переменные, не имеющие буквенного имени, вынесены вперед.)

Чтобы не повторяться, мы начинаем описание каждой переменной с одного из следующих обозначений:

Обозначение	Смысл
XXX	Устарела, *не используйте* в новом коде.
NOT	Not Officially There (только для внутреннего использования).
ALL	Подлинно глобальная, используется всеми пакетами.
PKG	Глобальная для пакета; в каждом пакете может быть своя.
FHA	Атрибут указателя файла; один на каждый объект ввода-вывода.
DYN	Автоматическая динамическая область видимости (влечет ALL).
LEX	Лексическая область видимости на этапе компиляции.
RO	Только для чтения, создает исключительную ситуацию при модификации.

Если перечислены несколько имен переменных или символов, то по умолчанию используется только короткое имя. Применение модуля `English` делает доступными длинные синонимы и только в текущем пакете, даже если переменная помечена обозначением [ALL].

Записи типа *`method HANDLE EXPR`* показывают объектно-ориентированные интерфейсы к переменным для указателей, предоставляемым модулем `FileHandle` и различными модулями `IO::`. (При желании можно использовать обозначения *`HANDLE->method (EXPR)`*.) Они позволяют избежать необходимости вызывать `select` для смены указателя вывода по умолчанию при просмотре или изменении этой переменной. Все такие методы возвращают прежнее значение атрибута `FileHandle`; новое значение присваивается, если задан аргумент *`EXPR`*. Если он не задан, то большинство методов ничего не делает с текущим значением, за исключением `autoflush`, который принимает аргумент `1` просто для того, чтобы чем-то выделиться.

_ (подчеркивание)

[ALL] Это специальный указатель файла, используемый для кэширования информации последнего успешного оператора `stat`, `lstat` или проверки файла (вроде `-w $file` или `-d $file`).

$цифры

[DYN,RO] Нумерованные переменные `$1`, `$2` и т. д. (до нужной вам величины)[1] содержат текст, соответствующий очередной паре круглых скобок последнего найденного шаблона в активной в данный момент динамической области видимости. (Мнемоника: как *\цифры*.)

[1] Хотя многие механизмы регулярных выражений поддерживают лишь до девяти ссылок на найденный текст, в Perl такого ограничения нет, поэтому если написать `$768`, Perl не станет возражать, в отличие от тех, кто будет сопровождать этот код, если в регулярном выражении действительно использовано столько скобок.

$[[XXX,LEX] Индекс первого элемента в массиве и первого символа в подстроке. По умолчанию 0, но мы устанавливали его в 1, чтобы Perl был более похож на *awk* (или FORTRAN) при индексации и вычислении функций index и substr. Поскольку присвоение $[было сочтено очень опасным, оно рассматривается теперь как директива компилятора с лексической видимостью и не может повлиять на какие-либо другие файлы. (Мнемоника: [начинает индексы.)

$# [XXX,ALL] Не применяйте ее, используйте вместо нее printf. $# содержит формат вывода чисел через print, что было несмелой попыткой эмуляции переменной OFMT из *awk*. (Мнемоника: # является знаком номера, но если вы умны (sharp)[1], то забудьте об этом, чтобы не превратить свою программу в мешанину и не быть за это наказанным.)

$* [XXX,ALL] Уф, три устаревших переменных подряд! Эту переменную можно (но не нужно) установить в истинное значение, чтобы заставить Perl предполагать /m в каждом шаблоне, в котором отсутствует явное /s. (Мнемоника: * соответствует множеству вещей.)

$a [PKG] Эта переменная используется в функции sort для первого из пары сравниваемых значений ($b содержит второе значение каждой пары). Пакет, к которому принадлежит $a, тот же, в который компилировался оператор sort, и не обязательно тот же, в который компилировалась его функция сравнения. Эта переменная неявно локализована в блоке сравнения sort. Так как она глобальная, то не вызывает претензий use strict. Поскольку это псевдоним для фактического массива, то может показаться, что его можно модифицировать, но делать этого не следует. См. описание sort.

$ACCUMULATOR

$^A

[ALL] Текущее значение накопителя write для строк format. Формат содержит команды formline, которые помещают свой результат в $^A. После вызова своего формата write выводит содержимое $^A и очищает ее. Поэтому вы никогда фактически не видите содержимого $^A, если только не вызовете formline самостоятельно, а потом не посмотрите на него. См. описание функции formline.

$ARG

$_ [ALL] Пространство по умолчанию для ввода и поиска по шаблону. Следующие пары эквивалентны:

```
while (<>) {...}    # эквивалентны только в обычной проверке while
while (defined($_ = <>)) {...}

chomp
chomp($_)
```

[1] Игра слов – «sharp» в переводе также означает «диез» (знак #). – *Примеч. перев.*

```
/^Subject:/
$_ =~ /^Subject:/

tr/a-z/A-Z/
$_ =~ tr/a-z/A-Z/
```

Вот те места, в которых Perl предполагает применение $_, если не указывать, над чем должны производиться действия:

- Списковые функции, например, print, unlink и такие унарные функции, как ord, pos и int, а также все проверки файлов (кроме -t, которое по умолчанию проводится с STDIN). Все функции, работающие по умолчанию с $_, отмечены также в главе 29 «Функции».
- Операции поиска по шаблону m// и s///, а также операции транслитерации y/// и tr///, когда они используются без оператора =~.
- Переменная итерации в цикле foreach (даже в виде for или если выступает в качестве модификатора команды), если не задана другая переменная.
- Неявная переменная итерации в функциях grep и map. (Задать для них другую переменную невозможно.)
- Место по умолчанию для входной записи, когда результат операций <FH>, readline или glob проверяется в качестве единственного критерия для проверки while. Это присвоение не производится вне проверки while или когда в выражение while включены дополнительные элементы.

(Мнемоника: подчеркивание в некоторых операциях подчеркивает используемый операнд.)

@ARG

@_ [ALL] Внутри подпрограммы этот массив содержит список аргументов, переданных подпрограмме. См. главу 6 «Подпрограммы». В скалярном контексте split помещает результат в этот массив, но такое применение осуждается.

ARGV

[ALL] Специальный указатель файла, который обходит все имена файлов в командной строке из @ARGV. Обычно записывается как нулевой указатель файла в операторе угловых скобок: <>.

$ARGV

[ALL] Содержит имя текущего файла при чтении из указателя ARGV в случае применения операторов <> или readline.

@ARGV

[ALL] Массив, содержащий аргументы командной строки, предназначенные для сценария. Заметьте, что $#ARGV обычно равно числу аргументов минус один, т. к. $ARGV[0] является первым аргументом, а не именем команды; используйте scalar @ARGV для получения количества аргументов программы. Имя программы можно найти в $0.

ARGVOUT

[ALL] Специальный указатель файла используется при обработке указателя ARGV с ключом -i или переменной $^I. См. описание ключа -i в главе 19 «Интерфейс командной строки».

$b [PKG] Эта переменная, товарищ $a, используется в сравнениях sort. Подробности см. в описании $a и функции sort.

$BASETIME

$^T

[ALL] Момент времени, в который начал выполняться сценарий, в секундах после начала эпохи (для систем Unix – начало 1970 года). Значения, возвращаемые проверками файлов -M, -A и -C, вычисляются относительно этого момента времени.

$CHILD_ERROR

$? [ALL] Статус, возвращенный последним закрытием канала, командой `` (обратного штриха) или функциями wait, waitpid или system. Обратите внимание, что это не просто код завершения, а все 16-разрядное слово, возвращаемое лежащими в основе системными вызовами *wait*(2) или *waitpid*(2) (или эквивалентными). Так, значение завершения подпроцесса находится в старшем байте, т. е. $? >> 8; в младшем байте $? & 127 сообщает о том, какой сигнал (если он был) послужил причиной завершения процесса, а $? & 128 сообщает о том, последовал ли за его кончиной дамп памяти. (Мнемоника: аналогична $? в *sh* и ее потомках.)

Внутри блока END переменная $? содержит значение, которое должно быть передано exit. Можно модифицировать $? в END, чтобы изменить статус завершения сценария.

В VMS прагма use vmsish 'status' заставляет отразить в $? подлинный статус выхода VMS вместо эмуляции по умолчанию статуса POSIX.

Если переменная h_errno поддерживается в C, ее числовое значение возвращается через $?, если неуспехом заканчивается вызов функций gethost*().

$COMPILING

$^C

[ALL] Текущее состояние внутреннего флага, связанное с ключом -c, в основном используемого с -MO и средством *perlcc*(1), чтобы позволить коду изменить свое поведение при компиляции для генерации кода. Например, вы, возможно, пожелаете выполнить AUTOLOAD на этапе компиляции вместо использования обычной отложенной загрузки, чтобы незамедлительно генерировать код. См. главу 18 «Компиляция».

DATA

[PKG] Этот специальный указатель файла ссылается на все, что расположено в текущем файле за маркерами __END__ или __DATA__. Маркер __END__ всегда открывает указатель файла main::DATA и так используется в глав-

ной программе. Маркер __DATA__ открывает указатель DATA в том пакете, в котором действует в данное время, поэтому различные модули могут иметь собственные указатели файла DATA, если они (предположительно) имеют различные имена пакетов.

$DEBUGGING

$^D

[ALL] Текущее значение внутренних флагов отладки, установленных согласно ключу командной строки -D; значения разрядов см. в главе 19 «Интерфейс командной строки». (Мнемоника: значение ключа -D.)

$EFFECTIVE_GROUP_ID

$EGID

$) [ALL] Текущий GID (ID группы) данного процесса. Если платформа поддерживает одновременное членство в нескольких группах, то $) позволяет получить разделенный пробелами список групп, в которых вы имеете членство. Первое число равно значению, возвращаемому *getegid*(2), а последующие – числам, возвращаемым *getgroups*(2), одно из которых может совпадать с первым.

Точно так же, значение, присваиваемое $), должно быть разделяемым пробелами списком чисел. Первое число используется для установки текущего GID, а остальные (если они имеются) передаются системному вызову *setgroups*(2). Чтобы задать для setgroups пустой список, просто повторите новый текущий GID; например, чтобы задать текущий GID равным 5 и указать пустой список для setgroups, скажите:

```
$) = "5 5";
```

(Мнемоника: круглые скобки применяются для *группировки* объектов. Текущий GID – это ваша группа, если вы выполняете setgid.) Заметьте: $<, $>, $(и $) могут устанавливаться только на машинах, поддерживающих соответствующую системную процедуру set-id. $(и $) можно менять местами только на машинах, поддерживающих *setregid*(2).

$EFFECTIVE_USER_ID

$EUID

$> [ALL] Текущий UID данного процесса, возвращаемый системным вызовом *geteuid*(2). Пример:

```
$< = $>;            # присвоить реальный uid текущему
($<,$>) = ($>,$<);  # поменять местами реальный и текущий uid
```

(Мнемоника: это UID, на который вы перешли при выполнении setuid.) Примечание: $< и $> можно менять местами только на машинах, поддерживающих *setreuid*(2). И то не всегда.

%ENV

[ALL] Хеш, в котором содержатся текущие переменные окружения. Присвоение значения %ENV изменяет окружение как вашего процесса, так и до-

черних процессов, запускаемых после присвоения. (Оно не может изменить окружение родительского процесса в любой системе, сходной с Unix.)

```
$ENV{PATH}  = "/bin:/usr/bin";
$ENV{PAGER} = "less";
$ENV{LESS}  = "MQeicsnf";  # наши любимые ключи для less(1)
system "man perl";         # получает новые установки
```

Для удаления чего-либо из окружения используйте функцию `delete`, а не `undef` со значением хеша.

Обратите внимание, что процессы, выполняющиеся как записи *crontab*(5), наследуют особенно обедненный набор переменных окружения. (Если программа прекрасно выполняется из командной строки, но не под *cron*, то, возможно, причина в этом.) Заметьте также, что необходимо установить `$ENV{PATH}`, `$ENV{SHELL}`, `$ENV{BASH_ENV}` и `$ENV{IFS}`, если сценарий выполняется как setuid. См. главу 23 «Защита данных».

`$EVAL_ERROR`

`$@` [ALL] Исключительная ситуация или сообщение о синтаксической ошибке Perl в последней операции `eval`. (Мнемоника: в каком месте («at») была синтаксическая ошибка?) В отличие от `$!` (`$OS_ERROR`), которая устанавливается при неудаче, но не очищается при успехе, `$@` всегда устанавливается (в значение «истина»), если последняя `eval` дала ошибку компиляции или исключительную ситуацию во время выполнения, и всегда сбрасывается (в значение «ложь»), если таких проблем не возникло.

Предупредительные сообщения в эту переменную не заносятся. Однако можно определить процедуру обработки предупреждений, установив `$SIG{__WARN__}`, как описывается далее в этом разделе.

Заметьте, что значением `$@` может быть не строка, а объект исключительной ситуации. Даже в этом случае с ним, вероятно, можно работать как со строкой, если в классе объекта исключительной ситуации определена перегрузка преобразования в строку. Если для распространения исключительной ситуации применятся команда:

```
die if $@;
```

то объект исключительной ситуации вызовет `$@->PROPAGATE`, чтобы решить, что делать. (Строковая исключительная ситуация просто добавляет к строке «propagated at».)

`$EXCEPTIONS_BEING_CAUGHT`

`$^S`

[ALL] Эта переменная отражает текущее состояние интерпретатора, возвращая «истину» при нахождении внутри `eval` и «ложь» в противном случае. Она не определена, если разбор текущей единицы компиляции еще не завершен, что может происходить в обработчиках `$SIG{__DIE__}` и `$SIG{__WARN__}`. (Мнемоника: state of `eval`.)

$EXECUTABLE_NAME

$^X

[ALL] Имя, с которым выполняется сам двоичный модуль *perl*, взятое из argv[0] в C (см. также описание переменной $PROGRAM_NAME ниже в этой главе).

@EXPORT

[PKG] К этой переменной массива обращается метод import модуля Exporter при поиске списка других переменных и подпрограмм пакета, которые нужно экспортировать по умолчанию при загрузке модуля по use или в случае применения тега импорта :DEFAULT. Эта переменная не является исключением при use strict, поэтому ее нужно объявлять как our или полностью квалифицированной именем пакета, если используется эта прагма. Однако все переменные, имена которых начинаются со строки «EXPORT», исключаются из предупреждений об однократном применении. См. главу 11 «Модули».

@EXPORT_OK

[PKG] К этой переменной массива обращается метод import модуля Exporter, чтобы определить, является ли законным запрашиваемый импорт. Она не является исключением при use strict. См. главу 11 «Модули».

%EXPORT_TAGS

[PKG] К этой переменной хеша обращается метод import модуля Exporter, когда запрашивается символ импорта с ведущим двоеточием, как в use POSIX ":sys_wait_h". Ключами являются теги двоеточия без ведущего двоеточия. Значениями должны быть ссылки на массивы, содержащие символы, которые нужно импортировать, когда запрашивается тег с двоеточием, которые все должны также присутствовать в @EXPORT или @EXPORT_OK. Переменная не является исключением при use strict. См. главу 11 «Модули».

$EXTENDED_OS_ERROR

$^E

[ALL] Информация об ошибке, специфичная для используемой операционной системы. В Unix переменная $^E идентична $! ($OS_ERROR), но отличается от нее в OS/2, VMS, системах Microsoft и в MacPerl. Специфическую информацию можно найти в документации по конкретной версии Perl для той или иной платформы. Предостережения, упомянутые в описании $!, обычно применимы к $^E. (Мнемоника: дополнительное толкование ошибок.)

@F

[PKG] Массив, в который помещаются поля расщепленной строки ввода, если задан ключ командной строки -a. Если данный ключ не указан, этот массив не имеет специального значения (и фактически представляет собой лишь @main::F, а не массив во всех пакетах).

%FIELDS

[NOT,PKG] Этот хеш предназначен для внутреннего использования в прагме use fields для определения текущих законных полей в хеше объекта. См. use fields, use base и раздел «Объявление полей с помощью use fields» главы 12 «Объекты».

format_formfeed *HANDLE EXPR*

$FORMAT_FORMFEED

$^L

[ALL] То, что неявно выводит функция write для подачи листа перед выводом заголовка формы. Значение по умолчанию равно "\f".

format_lines_left *HANDLE EXPR*

$FORMAT_LINES_LEFT

$- [FHA] Число строк, оставшихся на странице для выбранного в данный момент указателя файла вывода, для использования в объявлении format и функции write. (Мнемоника: строк_на_странице - строк_напечатано.)

format_lines_per_page *HANDLE EXPR*

$FORMAT_LINES_PER_PAGE

$= [FHA] Текущая длина страницы (количество строк для печати), соответствующая выбранному в данный момент указателю файла вывода для использования в format и функции write. По умолчанию равна 60. (Мнемоника: в = горизонтальные линии.)

format_line_break_characters *HANDLE EXPR*

$FORMAT_LINE_BREAK_CHARACTERS

$: [ALL] Текущий набор символов, после которых строка может быть разбита для заполнения полей продолжения (начинающихся с ^) в формате. Значение по умолчанию " \n-", что означает разбиение по пробельным символам и дефисам. (Мнемоника: colon (двоеточие) является техническим термином, означающим в поэзии часть строчки. Теперь нужно только запомнить мнемонику...)

format_name *HANDLE EXPR*

$FORMAT_NAME

$~ [FHA] Имя текущего формата отчета для текущего указателя выходного файла. Значением по умолчанию является имя указателя файла. (Мнемоника: поворот после $^.)

format_page_number *HANDLE EXPR*

$FORMAT_PAGE_NUMBER

$% [FHA] Номер текущей страницы для текущего указателя выходного файла, для использования с format и write. (Мнемоника: % служит регистром номера страницы в *troff*(1). Как, вы не знаете, что такое *troff*?)

format_top_name *HANDLE EXPR*

$FORMAT_TOP_NAME

$^ [FHA] Имя текущего формата верхнего колонтитула для текущего указателя выходного файла. По умолчанию представляет собой указатель файла, дополненный _TOP. (Мнемоника: указывает на верх страницы.)

$^H

[NOT,LEX] Эта переменная содержит разряды статуса для компилятора Perl (известные также, как «подсказки» (hints)) с лексической областью видимости. Эта переменная предназначена исключительно для внутреннего использования. Ее наличие, действие и содержимое могут изменяться без объявления. Тот, кто до нее дотронется, несомненно, умрет ужасной смертью от какой-нибудь отвратительной тропической болезни, неизвестной науке. (Мнемоника: знаем, но не скажем.)

%^H

[NOT,LEX] Хеш %^H обеспечивает такую же семантику лексической видимости, как и $^H, что делает его полезным для реализации прагм с лексической областью видимости. Прочтите зловещие предупреждения, относящиеся к $^H, и добавьте к ним то обстоятельство, что данная переменная все еще является экспериментальной.

%INC

[ALL] Хеш, который содержит имена всех файлов, загруженных Perl посредством do *FILE*, require или use. Ключами являются имена файлов, а значениями – фактические местонахождения файлов. Оператор require использует этот массив, чтобы определить, не загружен ли уже указанный файл. Например:

```
% perl -MLWP::Simple -le 'print $INC{"LWP/Simple.pm"}'
/opt/perl/5.6.0/lib/site_perl/LWP/Simple.pm
```

@INC

[ALL] Массив, содержащий список каталогов, где do *FILE*, require или use могут искать модули Perl. Первоначально состоит из аргументов ключа командной строки -I и каталогов в переменной окружения PERL5LIB, за которыми следуют библиотеки Perl по умолчанию, например:

```
/usr/local/lib/perl5/5.6.0/sun4-solaris
/usr/local/lib/perl5/5.6.0
/usr/local/lib/perl5/site_perl/5.6.0/sun4-solaris
/usr/local/lib/perl5/site_perl/5.6.0
/usr/local/lib/perl5/site_perl/5.00552/sun4-solaris
/usr/local/lib/perl5/site_perl/5.00552
/usr/local/lib/perl5/site_perl/5.005/sun4-solaris
/usr/local/lib/perl5/site_perl/5.005
/usr/local/lib/perl5/site_perl
```

за которыми следует символ «.», что указывает на текущий каталог. Для того чтобы изменить этот список из программы, включите прагму use lib, которая не только модифицирует переменную на этапе компиляции, но также добавляет некоторые каталоги, определяемые архитектурой (например, те, которые содержат библиотеки совместного доступа, используемые модулями XS):

```
use lib "/mypath/libdir/";
use SomeMod;
```

$INPLACE_EDIT

$^I

[ALL] Текущее значение расширения для редактирования по месту. Отключить редактирование по месту можно при помощи функции undef. К этой переменной можно обратиться из программы для получения такого же результата, как при использовании ключа -i. Например, чтобы получить эквивалент команды:

```
% perl -i.orig -pe 's/foo/bar/g' *.c
```

можно выполнить в программе такой код:

```
local $^I   = '.orig';
local @ARGV = glob("*.c");
while (<>) {
    s/foo/bar/g;
    print;
}
```

(Мнемоника: значение ключа -i.)

$INPUT_LINE_NUMBER

$NR

$. [ALL] Номер текущей записи (обычно номер строки) последнего указателя файла, из которого производилось чтение (или с которым выполнялись функции seek или tell). Значение может отличаться от фактического физического номера строки в файле – в зависимости от того, как определено текущее понятие «строки», – об управлении им см. описание $/ ($INPUT_RECORD_SEPARATOR). Явное выполнение close с указателем файла сбрасывает номер строки. Поскольку <> не выполняет close явно, строки имеют сквозную нумерацию в файлах ARGV (но см. примеры для eof). Локализация $. локализует и представление Perl о «последнем указателе файла, из которого производилось чтение». (Мнемоника: во многих программах «.» используется в качестве номера текущей строки.)

$INPUT_RECORD_SEPARATOR

$RS

$/ [ALL] Разделитель входных записей, по умолчанию – символ перевода строки, который применяется функциями readline, оператором <FH> и

функцией chomp. Действует подобно переменной RS в *awk* и при установке в нулевую строку рассматривает одну или более пустых строк как признак конца записи. (Пустая строка при этом не должна содержать пробелы или табуляции.) Можно установить значение из нескольких символов, и тогда будет отыскиваться многосимвольное окончание строки, но нельзя задавать шаблон – должен же *awk* быть в чем-то лучше.

Заметьте, что установка $/ в "\n\n" имеет смысл, несколько отличающийся от установки в "", если файл содержит пустые строки, располагающиеся подряд. При установке "" две или *более* последовательные пустые строки рассматриваются как одна пустая строка. Установка в "\n\n" означает, что Perl будет рассматривать третий перевод строки как принадлежащий новому абзацу.

Если сделать $/ неопределенной, то очередная операция ввода строки «проглотит» остаток файла как одну скалярную величину:

```
undef $/;           # enable whole-file mode
$_ = <FH>;          # whole file now here
s/\n[ \t]+/ /g;     # fold indented lines
```

Если конструкция while (<>) применяется для доступа к указателю ARGV при неопределенной $/, то каждая операция чтения вводит очередной файл:

```
undef $/;
while (<>) {        # $_ содержит очередной файл целиком
    ...
}
```

Выше была использована undef, но безопаснее сделать $/ неопределенной с помощью local:

```
{
    local $/;
    $_ = <FH>;
}
```

Присвоение $/ ссылки на целое число, скаляра, содержащего целое число, или скаляра, который можно преобразовать в целое число, заставит операции readline и <FH> считывать записи фиксированной длины (при этом максимальным размером записи будет это целое число) вместо записей переменной длины, оканчивающихся указанной строкой. Поэтому код:

```
$/ = \32768; # или \"32768" или \$scalar_var_containing_32768
open(FILE, $myfile);
$record = <FILE>;
```

будет читать запись не длиннее 32 768 байт из указателя FILE. Если чтение выполняется не из файла, ориентированного на записи (или операционная система не поддерживает файлы, ориентированные на записи), то,

вероятно, при каждом чтении будет получен полный набор данных. Если запись длиннее установленного размера, то вы получите запись частями. Режим записей можно смешивать с построчным режимом только в тех системах, в которых стандартный ввод/вывод предоставляет функцию *read*(3); известным исключением является VMS.

Вызов `chomp`, когда установка `$/` активизирует режим записей или не определена, не оказывает эффекта. См. также ключи командной строки `-0` (цифра) и `-l` (буква) в главе 19 «Интерфейс командной строки». (Мнемоника: / используется в качестве разделителя строк при цитировании стихотворений.)

`@ISA`

[PKG] Этот массив содержит имена других пакетов, которые нужно просматривать, если вызываемый метод отсутствует в текущем пакете. Это значит, что он содержит базовые классы пакета. Он неявно устанавливается директивой `use base`. Он не исключается из сообщений `use strict`. См. главу 12.

`@LAST_MATCH_END`

`@+` [DYN,RO] Данный массив содержит смещения концов последних успешных вложенных совпадений в активной в данный момент динамической области видимости. `$+[0]` представляет собой смещение конца совпадения в целом. Это то же самое значение, которое возвращает функция `pos` для переменной, в которой проводился поиск. (Говоря «смещение конца», мы в действительности имеем в виду смещение первого символа, *следующего* за концом найденного совпадения, чтобы можно было вычитать смещения начал из смещений концов и получать длину.) *n*-й элемент этого массива содержит смещение *n*-го вложенного соответствия, поэтому `$+[1]` является смещением конца `$1`, `$+[2]` – смещением конца `$2` и т. д. С помощью `$#+` можно определить, сколько подгрупп было в последнем успешном поиске. См. также `@-` (`@LAST_MATCH_START`).

После успешного поиска в некоторой переменной `$var`:

- `$` ` то же самое, что `substr($var, 0, $-[0])`
- `$&` то же самое, что `substr($var, $-[0], $+[0] - $-[0])`
- `$'` то же самое, что `substr($var, $+[0])`
- `$1` то же самое, что `substr($var, $-[1], $+[1] - $-[1])`
- `$2` то же самое, что `substr($var, $-[2], $+[2] - $-[2])`
- `$3` то же самое, что `substr($var, $-[3], $+[3] - $-[3])` и т. д.

`@LAST_MATCH_START`

`@-` [DYN,RO] Данный массив содержит смещения начал последних успешных вложенных совпадений в активной в данный момент динамической области видимости. `$-[0]` является смещением начала совпадения в целом. *n*-й элемент этого массива содержит смещение *n*-го вложенного соответствия, поэтому `$-[1]` равно смещению начала `$1`, `$-[2]` – смещением

начала $2 и т. д. С помощью $#- можно определить, сколько подгрупп было в последнем успешном поиске. См. также @+ (@LAST_MATCH_END).

$LAST_PAREN_MATCH

$+ [DYN,RO] Эта переменная возвращает последнее вложенное совпадение в скобках для последнего найденного шаблона в текущей динамической области видимости. Это полезно, когда неизвестно (или безразлично), который из группы альтернативных шаблонов был найден. (Мнемоника: будь решительным и смотри вперед.) Пример:

```
$rev = $+   if /Version: (.*)|Revision: (.*)/;
```

$LAST_REGEXP_CODE_RESULT

$^R

[DYN] Эта переменная содержит результат последнего фрагмента кода, выполненного внутри успешного поиска по шаблону, в конструкции (?{ *CODE* }). $^R предоставляет возможность выполнить код и запомнить его результат для использования далее в шаблоне или еще позже.

По мере передвижения механизма регулярных выражений Perl по шаблону ему могут встретиться несколько выражений (?{ *CODE* }). Поэтому он запоминает каждое значение $^R, и когда позже ему приходится осуществлять возврат, он восстанавливает прежнее значение $^R. Иными словами, $^R имеет в шаблоне динамическую область видимости, аналогично $1 и прочим.

Поэтому $^R представляет не просто результат последнего фрагмента кода, выполненного в шаблоне. Это результат последнего фрагмента кода, *ведущий к успешному поиску соответствия*. Следствием является восстановление предшествующего поиску значения $^R при неудачном поиске.

Если шаблон (?{ *CODE* }) действует непосредственно как условие подшаблона (?(*COND*) *IFTRUE*|*IFFALSE*), то значение $^R не устанавливается.

$LIST_SEPARATOR

$" [ALL] Когда массив или его срез интерполируются в строку в двойных кавычках (или аналогичную конструкцию), эта переменная задает строку, помещаемую между отдельными элементами. По умолчанию равна пробелу. (Мнемоника: надо думать, очевидная.)

$^M

[ALL] По умолчанию ошибка нехватки памяти не перехватывается. Однако если при компиляции *perl* была зарезервирована возможность применения $^M, то ей можно отвести роль аварийного пула памяти. Если Perl скомпилирован с -DPERL_EMERGENCY_SBRK и использует malloc Perl, то:

```
$^M = 'a' x (1 << 16);
```

выделит буфер 64 Кбайт для аварийного использования. О том, как включить эту опцию, прочтите в файле *INSTALL* в каталоге дистрибутива исходного кода Perl. Чтобы не поощрять случайное применение этой разви-

той функции, для этой переменной не определено длинное имя (а какая мнемоника, мы вам не скажем).

$MATCH

$& [DYN,RO] Строка, найденная при последнем успешном поиске по шаблону в активной в данный момент динамической области видимости. (Мнемоника: как & в некоторых редакторах.)

$OLD_PERL_VERSION

$] [ALL] Возвращает номер версии + уровень патча (patchlevel)/1000. Может использоваться в начале сценария для определения пригодности имеющейся версии интерпретатора Perl. (Мнемоника: эта версия Perl в правильных рамках?) Пример:

```
warn "No checksumming!\n" if $] < 3.019;
die "Must have prototyping available\n" if $] < 5.003;
```

См. также в описании директив use *VERSION* и require *VERSION* об удобных способах завершения работы, если версия интерпретатора Perl слишком старая. См. описание $^V как более гибкого представления версии Perl в UTF-8.

$OSNAME

$^O

[ALL] Эта переменная содержит название платформы (обычно операционной системы), для которой был скомпилирован текущий двоичный модуль *perl*. Это недорогая альтернатива вытаскиванию ее из модуля Config.

$OS_ERROR

$ERRNO

$! [ALL] При использовании в числовом контексте выдает текущее значение последней ошибки системного вызова при действии всех обычных предостережений. (Это значит, что не следует полагаться на какие-либо конкретные значения $!, если только не возвращено особое значение, указывающее на системную ошибку.) В случае применения в строковом контексте $! выдает соответствующее сообщение о системной ошибке. Можно присвоить $! номер ошибки, если, например, требуется, чтобы $! возвратила строку, соответствующую этой ошибке, или чтобы установить значение выхода для die. См также описание модуля Errno в главе 32. (Мнемоника: что там грохнулось?)

%OS_ERROR

%ERRNO

%! [ALL] Этот хеш определен, только если был загружен стандартный модуль Errno, описанный в главе 32. Если это так, то можно указывать в качестве индекса %! конкретное описание ошибки и получаемое значение будет истинным, только если это текущая ошибка. Например, $!{ENOENT} истинно, только если переменная C errno в настоящее время имеет значе-

ние C #define, ENOENT. Это удобно для доступа к символам из специфических наборов поставщиков.

autoflush *HANDLE EXPR*

$OUTPUT_AUTOFLUSH

$AUTOFLUSH

$| [FHA] Будучи установлена в истинное значение, вызывает очистку буфера после каждой команды print, printf и write для выбранного в данный момент указателя файла вывода. (Мы называем это *буферизацией команд*. В противоположность распространенному мнению, установка этой переменной не отключает буферизацию.) Значением по умолчанию является «ложь», что для многих систем означает, что STDOUT будет буферизоваться построчно, если вывод происходит на терминал, и буферизоваться поблочно в ином случае, даже для каналов и сокетов. Установка этой переменной полезна, если вывод производится в канал, например, если запускается сценарий Perl под *rsh*(1) и требуется наблюдать за выводом по мере его выполнения. Если в выходном буфере текущего выбранного указателя файла есть несброшенные данные, когда эта переменная устанавливается в истинное значение, то в качестве побочного эффекта этот буфер будет немедленно очищен. Примеры управления буферизацией для указателей файлов, отличных от STDOUT см. в описании формата select с одним аргументом. (Мнемоника: когда вам нужно, чтобы каналы качали «от души».)

Эта переменная не оказывает влияния на входную буферизацию, о которой сказано в описании функции getc в главе 29 или примере для модуля POSIX в главе 32.

$OUTPUT_FIELD_SEPARATOR

$OFS

$, [ALL] Разделитель полей вывода (фактически указатель конца) для print. Обычно print просто выводит указанный список элементов, между которыми ничего нет. Устанавливайте эту переменную так же, как переменную *awk* OFS для задания символов, выводимых между полями. (Мнемоника: то, что выводится, когда в команде print есть «,».)

$OUTPUT_RECORD_SEPARATOR

$ORS

$\ [ALL] Разделитель выходных записей (фактически указатель конца) для print. Обычно print просто выводит через запятую указанные поля без символа перевода строки или разделителя записей в конце. Устанавливайте эту переменную так же, как переменную *awk* ORS для задания символов, выводимых в конце print. (Мнемоника: $\ устанавливается вместо "\n", добавляемого в конце вывода. Кроме того, она похожа на /, но это то, что мы получаем «обратно» от Perl.) См. также описание ключа командной строки -l (от «line») в главе 19.

%OVERLOAD

[NOT,PKG] Элементы этого хеша устанавливаются внутренне для использования директивы overload с целью реализации перегрузки операторов для объектов класса текущего пакета. См. главу 13 «Перегрузка».

$PERLDB

$^P

[NOT,ALL] Внутренняя переменная для включения отладчика Perl (*perl-d*).

$PERL_VERSION

$^V

[ALL] Пересмотр, версия и подверсия интепретатора Perl, представленная как двоичная «строка версии». V-строки, вообще-то, не имеют числового значения, но переменная $^V имеет двойное значение, при этом числовое эквивалентно старой переменной $], т. е. представляет собой действительное число, высчитываемое как ревизия + версия/1000 + подверсия/1 000 000. Строковое значение составлено из символов UTF-8: chr($revision) . chr($version) . chr($subversion). Это значит, что $^V *не* печатается. Для вывода ее нужно сказать:

```
printf "%vd", $^V;
```

Положительной стороной является то, что можно использовать обычное сравнение строк, чтобы определить, находится ли версия интерпретатора Perl, выполняющего сценарий, в диапазоне «правильных» значений. (Это относится к любым номерам версий, представляемым как v-строки, не только Perl.) Пример:

```
warn "Нет объявлений 'our'!\n" if $^V lt v5.6;
```

Читайте в описании use *VERSION* и require *VERSION* о том, как удобно завершить работу, если используемый интерпретатор Perl старше, чем вы рассчитывали. См. также в $], как изначально представлялась версия Perl.

$POSTMATCH

$' [DYN,RO] Строка, следующая за тем, что было найдено в последнем удачном поиске по шаблону в активной в данный момент динамической области видимости. (Мнемоника: ' часто следует за цитируемой строкой.) Пример:

```
$_ = 'abcdefghi';
/def/;
print "$`:$&:$'\n";          # выводит abc:def:ghi
```

Благодаря динамической области видимости Perl не может знать, каким шаблонам потребуется сохранять свои результаты в этих переменных, поэтому задание $` или $' где-либо в программе вызывает снижение производительности для всех поисков по шаблону в программе. В небольших программах это не вызывает особых проблем, но при написании кода

многократно используемого модуля применения этой пары следует, вероятно, избегать. Приведенный выше пример можно переработать, чтобы не наносить урона общей производительности:

```
$_ = 'abcdefghi';
/(.*?)(def)(.*)/s;          # /s на случай, если $1 содержит перевод строки
print "$1:$2:$3\n";         # выводит abc:def:ghi
```

$PREMATCH

$` [DYN,RO] Строка, предшествующая тому, что было найдено при последнем удачном поиске по шаблону в активной в данный момент динамической области видимости. (Мнемоника: ` часто предшествует цитируемой строке.) См. выше замечание о производительности для $`.

$PROCESS_ID

$PID

$$ [ALL] Номер процесса (PID) для Perl, выполняющего данный сценарий. Эта переменная автоматически обновляется при выполнении fork. На практике можно даже установить $$ самостоятельно; при этом, однако, PID процесса не изменится. Это ловкий фокус. (Мнемоника: та же, что в различных оболочках.)

Нужно проявлять осторожность в использовании $$, – она может быть неправильно интерпретирована как разыменование: $$alphanum. В такой ситуации пишите ${$}alphanum, чтобы отличить ее от ${$alphanum}.

$PROGRAM_NAME

$0 [ALL] Хранит имя файла, содержащего выполняющийся сценарий Perl. Присвоение $0 является магическим: оно пытается изменить область аргументов, о которой обычно сообщает программа *ps*(1). Это полезнее применять как способ индикации состояния текущей программы, чем для сокрытия выполняемой программы. Но это работает не на всех системах. (Мнемоника: то же, что в *sh*, *ksh*, *bash* и т. д.)

$REAL_GROUP_ID

$GID

$([ALL] Реальный ID группы (GID) данного процесса. Если платформа поддерживает одновременное членство в нескольких группах, то $(дает разделенный пробелами список групп, в которых вы имеете членство. Первое число – то, которое возвращает *getgid*(2), а последующие представляют собой числа, возвращаемые *getgroups*(2), одно из которых может совпадать с первым числом.

Однако значение, присваиваемое $(, должно быть одним числом, используемым для установки реального GID. Поэтому значение, которое дает $(, *не* должно присваиваться обратно $(, кроме как принудительно преобразовываясь в число, например, путем прибавления нуля. Это связано с тем, что реальная группа может быть только одна. См. описание $) ($EFFECTIVE_GROUP_ID), позволяющее устанавливать несколько текущих групп.

(Мнемоника: круглые скобки предназначены для *группировки* объектов. Реальный GID – это группа, которую мы покинули (*left*), выполняя setgid.)

$REAL_USER_ID

$UID

$< [ALL] Реальный ID пользователя (UID) данного процесса, как он возвращается системным вызовом *getuid*(2). Можно ли его изменить и как, зависит от особенностей реализации системы – см. примеры в $> ($EFFECTIVE_USER_ID). (Мнемоника: это UID, *от* которого мы пришли, выполняя setuid.)

%SIG

[ALL] Хеш, используемый для установки обработчиков различных сигналов. (См. раздел «Сигналы» главы 16 «Межпроцессное взаимодействие».) Например:

```
sub handler {
    my $sig = shift;   # 1-й аргумент является именем сигнала
    syswrite STDERR, "Перехват SIG$sig- -завершение работы\n";
                       # Избегайте использования стандартного ввода-вывода в
                       # асинхронных обработчиках, чтобы избежать дампов памяти.
                       # (Даже конкатенация строк опасна.)
    close LOG;         # Обращается к стандартному I/O, поэтому может быть дамп!
    exit 1;            # Но так как мы выходим, можно попробовать.
}

$SIG{INT}  = \&handler;
$SIG{QUIT} = \&handler;
...
$SIG{INT}  = 'DEFAULT';    # восстановить действие по умолчанию
$SIG{QUIT} = 'IGNORE';     # игнорировать SIGQUIT
```

Хеш %SIG содержит неопределенные значения, соответствующие сигналам, для которых не установлены обработчики. Обработчик можно задать как ссылку на подпрограмму или строку. Строковое значение, не являющееся одним из специальных действий "DEFAULT" или "IGNORE", представляет собой имя функции, которая, если ее имя не квалифицировано пакетом, полагается находящейся в пакете main. Вот еще несколько примеров:

```
$SIG{PIPE} = "Plumber";   # okay, предполагает main::Plumber
$SIG{PIPE} = \&Plumber;   # отлично, использовать Plumber из текущего пакета
```

С помощью хеша %SIG можно установить некоторые внутренние ловушки. Программа, на которую указывает $SIG{__WARN__}, вызывается перед тем, как должно быть выведено предупредительное сообщение. Сообщение передается в первом аргументе. Наличие ловушки __WARN__ вызывает подавление обычного вывода предупреждений на STDERR. Ее можно использовать для сохранения предупреждений в переменной или чтобы сделать предупреждения фатальными ошибками, например:

```
local $SIG{__WARN__} = sub { die $_[0] };
eval $proggie;
```

Это аналогично высказыванию:

```
use warnings qw/FATAL all/;
eval $proggie;
```

за исключением того, что в первом случае область видимости динамическая, а во втором – лексическая.

Программа, на которую указывает `$SIG{__DIE__}`, обеспечивает превращение исключительной ситуации-лягушки в исключительную ситуацию-принца с помощью волшебного поцелуя, что часто не работает. Наилучшее применение – для умирающей программы, которая должна скончаться от неперехваченной исключительной ситуации, чтобы произвести некую завершающую обработку перед выходом. От смерти это не спасет, но еще раз крикнуть «ура!» можно.

Сообщение исключительной ситуации передается в первом аргументе. При возврате из ловушки `__DIE__` обработка исключительной ситуации выполняется, как если бы ловушки и не было, если только программа-ловушка не выйдет сама через `goto`, выход из цикла или `die`. Во время вызова обработчик `__DIE__` явным образом отключается, чтобы вы сами могли вызвать настоящий `die` из обработчика `__DIE__`. (Если его не отключить, то обработчик будет рекурсивно вызывать себя до бесконечности.) Обработчик для `$SIG{__WARN__}` работает аналогичным образом.

Устанавливать `$SIG{__DIE__}` нужно только в основной программе, но не в модулях. Это связано с тем, что в настоящее время даже перехватываемые исключительные ситуации все же запускают обработчик `$SIG{__DIE__}`. Это весьма не одобряется, поскольку может нарушить работу невинных модулей, не ожидающих, что предполагаемые ими исключительные ситуации таинственным образом изменяются. Используйте эту возможность только в качестве последнего средства, и если приходится это делать, всегда ставьте впереди `local`, чтобы сократить опасный период.

Не пытайтесь основывать на этой возможности механизм обработки исключительных ситуаций. Вместо этого для перехвата исключительных ситуаций используйте `eval {}`.

`STDERR`

[ALL] Специальный указатель файла для стандартного устройства вывода ошибок во всех пакетах.

`STDIN`

[ALL] Специальный указатель файла для стандартного устройства ввода во всех пакетах.

`STDOUT`

[ALL] Специальный указатель файла для стандартного устройства вывода во всех пакетах.

$SUBSCRIPT_SEPARATOR

$SUBSEP

$; [ALL] Разделитель индексов для эмуляции многомерных хешей. Если вы ссылаетесь на элемент хеша как:

```
$foo{$a,$b,$c}
```

то в действительности это значит:

```
$foo{join($;, $a, $b, $c)}
```

Но не пишите:

```
@foo{$a,$b,$c}      # срез, обратите внимание на @
```

что означает:

```
($foo{$a},$foo{$b},$foo{$c})
```

Значением по умолчанию является "\034" – то же, что SUBSEP в *awk*. Обратите внимание, что если в ключах содержатся двоичные данные, то безопасного значения для $; может не найтись. (Мнемоника: запятая – разделитель синтаксических субскриптов – это половина точки с запятой. Коряво, конечно, но $, уже занята для более важных задач.)

Хотя мы не объявили эту функцию устаревшей, но сейчас лучше использовать «настоящие» многомерные хеши, т. е. $foo{$a}{$b}{$c}, вместо $foo{$a,$b,$c}. Однако может оказаться, что поддельные хеши легче сортируются и лучше поддаются использованию в качестве DBM-файлов.

$SYSTEM_FD_MAX

$^F

[ALL] Максимальный «системный» дескриптор файла, обычно 2. Системные дескрипторы файлов передаются новым программам во время exec, а более высокие дескрипторы – нет. Кроме того, во время выполнения open системные дескрипторы файлов сохраняются, даже если вызов этой функции завершается неудачей. (Обычные дескрипторы файлов закрываются перед попыткой выполнения open и остаются закрытыми при ее отказе.) Обратите внимание на то, что статус дескриптора файла в отношении закрытия при exec определяется согласно значению $^F во время выполнения open, а не во время выполнения exec. Этого можно избежать, сначала временно присвоив $^F заведомо большое значение:

```
{
    local $^F = 10_000;
    pipe(HITHER,THITHER) or die "can't pipe: $!";
}
```

$VERSION

[PKG] Обращение к этой переменной происходит, когда задается минимальная допустимая версия модуля, как в use SomeMod 2.5. Если $Some-

Mod::VERSION меньше этого значения, имеет место исключительная ситуация. Эту переменную рассматривает метод UNIVERSAL->VERSION, поэтому можно определить собственную функцию VERSION в текущем пакете, если вам нужно что-либо иное, кроме режима по умолчанию. См. главу 12.

$WARNING

$^W

[ALL] Текущее булево значение ключа глобального предупреждения (не путать с глобальным потеплением, относительно которого слышно много глобальных предупреждений).[1] См. также директиву use warnings в главе 31 «Модули прагм» и ключи командной строки -W и -X для предупреждений с лексической областью видимости, на которые данная переменная не влияет. (Мнемоника: значение связано с ключом -w.)

${^WARNING_BITS}

[NOT,ALL] Текущий набор проверок, активированных директивой use warnings. Подробности см. в описании use warnings в главе 31.

${^WIDE_SYSTEM_CALLS}

[ALL] Глобальный флаг, разрешающий всем системным вызовам, осуществляемым Perl, использовать API расширенных символов, родные для системы, если таковые имеются. Включить эту возможность можно из командной строки с помощью ключа -C. Начальное значение обычно 0 для совместимости с Perl версий до 5.6, но Perl может автоматически устанавливать его в 1, если система предоставляет устанавливаемое пользователем значение по умолчанию (например, через $ENV{LC_CTYPE}). Прагма use bytes всегда переопределяет действие этого флага в текущей лексической области видимости.

А теперь приготовьтесь к изучению *большой* главы...

[1] В оригинале обыгрывается созвучие выражений «global warning» и «global warming». – *Примеч. ред.*

29

Функции

В этой главе описываются встроенные функции Perl в алфавитном порядке,[1] обеспечивающем удобство получения справки. Описание каждой функции начинается с краткой сводки ее синтаксиса. Имена параметров, такие как *THIS*, представляют заместители фактических выражений, а текст, следующий за синтаксической сводкой, описывает семантику применения (или пропуска) фактических аргументов.

Можно представлять себе функции как термы в выражении, наряду с литералами и переменными. Либо можно представлять их себе как префиксные операторы, обрабатывающие аргументы, следующие за ними. В половине случаев мы так и называем их операторами.

Некоторые операторы (функции) принимают в качестве аргумента *LIST* (список). Элементы, входящие в *LIST*, должны разделяться запятыми (или `=>`, что является причудливой разновидностью запятой). Элементы списка вычисляются в списковом контексте, поэтому каждый элемент возвращает скалярное или списковое значение, в зависимости от своей чувствительности к списковому контексту. Каждое возвращаемое значение, скалярное или список, будет интерполировано как часть общей последовательности скалярных значений. Это значит, что все списки уплощаются в один список. С точки зрения функции, получающей аргументы, общий аргумент *LIST* всегда является одномерной списковой величиной. (Для интерполяции мас-

---

[1] Иногда тесно связанные между собой функции объединены на страницах руководства, и мы придерживаемся здесь такой же группировки. Например, чтобы найти описание `endpwent`, придется смотреть `getpwent`.

сива как одного элемента необходимо явно создать и интерполировать ссылку на массив.)

Предопределенные функции Perl могут быть использованы с круглыми скобками вокруг аргументов или без них; в синтаксических сводках данной главы скобки опущены. Если вы все же используете скобки, простое, но часто вызывающее удивление правило состоит в следующем: если нечто выглядит как функция, то это и *есть* функция, поэтому приоритет не имеет значения. В противном случае это списковый оператор или унарный оператор, и приоритет имеет значение. Будьте осторожны, поскольку даже если поместить пробельный символ между ключевым словом и левой скобкой, это все равно останется функцией:

```
print 1+2*4;        # Выводит 9.
print(1+2) * 4;     # Выводит 3!
print (1+2)*4;      # Тоже выводит 3!
print +(1+2)*4;     # Выводит 12.
print ((1+2)*4);    # Выводит 12.
```

Если запустить Perl с ключом `-w`, будет получено предупреждение. Например, вторая и третья строки из вышеприведенных порождают такие сообщения:

```
print (...) interpreted as function at - line 2.
[print (...) интерпретируется как функция - в строке 2]
Useless use of integer multiplication in void context at - line 2.
[Напрасное использование умножения целых в пустом контексте - в строке 2]
```

При условии простого определения некоторых функций программисту предоставляется значительная свобода в способе передачи аргументов. Например, самый распространенный способ применения `chmod` состоит в передаче прав доступа к файлу (режима) в качестве начального аргумента:

```
chmod 0644, @array;
```

но в определении `chmod` просто сказано:

```
chmod LIST
```

поэтому с таким же успехом можно сказать:

```
unshift @array, 0644;
chmod @array;
```

Если первый аргумент списка не является допустимым режимом, вызов `chmod` оказывается неудачным, но это семантическая проблема этапа исполнения, не связанная с синтаксисом вызова. Если семантика требует передачи вначале каких-либо специальных аргументов, то эти ограничения будут описаны в тексте.

В противоположность простым функциям *LIST*, другие функции налагают дополнительные синтаксические ограничения. Например, для push синтаксически правильным является такое написание :

```
push ARRAY, LIST
```

Это значит, что push требует действительный массив в качестве своего первого аргумента, но безразлична к следующим аргументам. Вот что означает *LIST* в конце. (Списки всегда идут в конце, т. к. они «съедают» все оставшиеся значения.) Если в списке параметров вызова есть аргументы, стоящие перед *LIST*, они синтаксически различаются компилятором, а не просто семантически различаются интерпретатором при его последующем выполнении. Такие аргументы никогда не вычисляются в списковом контексте. Они могут вычисляться в скалярном контексте или быть особыми ссылочными аргументами, как массив в push. (В описании сказано, кто является кем.)

Для тех операций, которые прямо основываются на функциях библиотеки C, мы не стали дублировать вашу системную документацию. Если в описании function предлагается посмотреть *function*(2), это значит, что нужно найти соответствующую C-версию этой функции, чтобы больше узнать о ее семантике. Число в скобках указывает на раздел в руководстве системного программиста, в котором вы найдете страницу руководства, если у вас установлено руководство (manpages). (И не найдете, если оно не установлено.)

На этих страницах руководства могут документироваться зависящие от системы функции, например, файлы паролей (*/etc/shadow*), списки контроля доступа и т. д. Многие функции Perl, являющиеся производными от библиотечных функций C в Unix, эмулируются даже на отличных от Unix платформах. Например, хотя операционная система может не поддерживать системные вызовы *flock*(2) или *fork*(2), Perl, тем не менее, постарается эмулировать их с использованием тех средств, которые предоставляет данная платформа.

Иногда обнаруживается, что документированная функция C принимает больше аргументов, чем соответствующая функция Perl. Обычно отсутствующие аргументы – это то, что Perl уже знает, например, длина предыдущего аргумента, поэтому передавать такие аргументы в Perl не требуется. Все остальные несоответствия обусловлены различным заданием указателей файлов в Perl и C, а также значений успеха/неуспеха.

В целом функции Perl, служащие оболочками одноименных системных вызовов (например, *chown*(2), *fork*(2), *closedir*(2) и т. д.), возвращают «истину» при успехе и undef в противном случае, что указывается в нижеприводимых описаниях. В этом состоит отличие от интерфейсов библиотеки C для этих операций, которые всегда возвращают -1 при неудаче. Исключениями из этого правила являются wait, waitpid и syscall. Системные вызовы при неудаче также устанавливают специальную переменную $! ($OS_ERROR). Прочие функции этого не делают, разве что случайно.

Для функций, которые могут использоваться как в скалярном, так и в списковом контекстах, неудача обычно указывается в скалярном контексте путем

возврата ложного значения (обычно `undef`) и в списковом контексте путем возврата нулевого списка. На успешное выполнение обычно указывает возврат значения, которое вычисляется как «истина» (в контексте).

Запомните следующее правило: *нет* правила, связывающего характер поведения функции в списковом контексте с ее поведением в скалярном контексте, и наоборот. Она может делать две совершенно разные вещи.

Каждая функция знает контекст, в котором она вызвана. Функция, которая возвращает список при вызове в списковом контексте, при вызове в скалярном контексте возвращает значение того типа, который наиболее уместен. Некоторые функции возвращают длину списка, который был бы возвращен в списковом контексте. Некоторые операторы возвращают первое значение в списке. Некоторые функции возвращают последнее значение в списке. Какие-то функции возвращают «другое» значение, когда нечто можно искать либо по номеру, либо по имени. Некоторые функции возвращают число успешных операций. В целом функции Perl выполняют в точности то, что вам нужно, если только вам не нужно единообразие.

Одно заключительное замечание: мы постарались быть очень последовательными в использовании терминов «байт» и «символ». Исторически эти термины всегда смешивались. Когда бы говорим «байт», то всегда подразумеваем октет, 8 бит. Когда мы говорим «символ», то подразумеваем абстрактный символ, обычно символ Unicode, который может быть представлен в строках одним или более байтами.

Но обратите внимание на это «обычно». Perl намеренно путает байты с символами в области видимости объявления `use bytes`, поэтому, когда мы говорим «символ», необходимо понимать это как байт в контексте `use bytes` и как символ Unicode в противном случае. Иными словами, `use bytes` просто искажает определение символа, делая его таким, каким оно было в старых версиях Perl. Поэтому, например, когда мы говорим, что скалярная функция `reverse` обращает порядок символов в строке, не надо задавать вопрос, к чему *в действительности* это относится – к символам или байтам, поскольку ответ будет «да».

Функции Perl по категориям

Вот функции Perl и ключевые слова, похожие на функции, разбитые на категории. Некоторые функции появляются более чем под одним заголовком.

Обработка скаляров

`chomp`, `chop`, `chr`, `crypt`, `hex`, `index`, `lc`, `lcfirst`, `length`, `oct`, `ord`, `pack`, `q//`, `qq//`, `reverse`, `rindex`, `sprintf`, `substr`, `tr///`, `uc`, `ucfirst`, `y///`

Регулярные выражения и поиск по шаблону

`m//` `pos`, `qr//`, `quotemeta`, `s///`, `split`, `study`

Числовые функции

`abs`, `atan2`, `cos`, `exp`, `hex`, `int`, `log`, `oct`, `rand`, `sin`, `sqrt`, `srand`

Обработка массивов

pop, push, shift, splice, unshift

Обработка списков

grep, join, map, qw//, reverse, sort, unpack

Обработка хешей

delete, each, exists, keys, values

Ввод и вывод

binmode, close, closedir, dbmclose, dbmopen, die, eof, fileno, flock, format, getc, print, printf, read, readdir, readpipe, rewinddir, seek, seekdir, select **(готовые указатели файлов)**, syscall, sysread, sysseek, syswrite, tell, telldir, truncate, warn, write

Данные фиксированной длины и записи

pack, read, syscall, sysread, sysseek, syswrite, unpack, vec

Указатели файлов, файлы и каталоги

chdir, chmod, chown, chroot, fcntl, glob, ioctl, link, lstat, mkdir, open, opendir, readlink, rename, rmdir, select **(готовые указатели файлов)**, select **(указатель выходного файла)**, stat, symlink, sysopen, umask, unlink, utime

Управление последовательностью выполнения команд

caller, continue, die, do, dump, eval, exit, goto, last, next, redo, return, sub, wantarray

Области видимости

caller, import, local, my, no, our, package, use

Разное

defined, dump, eval, formline, lock, prototype, reset, scalar, undef, wantarray

Процессы и группы процессов

alarm, exec, fork, getpgrp, getppid, getpriority, kill, pipe, qx//, setpgrp, setpriority, sleep, system, times, wait, waitpid

Библиотечные модули

do, import, no, package, require, use

Классы и объекты

bless, dbmclose, dbmopen, package, ref, tie, tied, untie, use

Низкоуровневый доступ к сокетам

accept, bind, connect, getpeername, getsockname, getsockopt, listen, recv, send, setsockopt, shutdown, socket, socketpair

Межпроцессное взаимодействие в System V

msgctl, msgget, msgrcv, msgsnd, semctl, semget, semop, shmctl, shmget, shmread, shmwrite

Получение информации о пользователях и группах

endgrent, endhostent, endnetent, endpwent, getgrent, getgrgid, getgrnam, getlogin, getpwent, getpwnam, getpwuid, setgrent, setpwent

Получение информации о сети

endprotoent, endservent, gethostbyaddr, gethostbyname, gethostent, getnetbyaddr, getnetbyname, getnetent, getprotobyname, getprotobynumber, getprotoent, getservbyname, getservbyport, getservent, sethostent, setnetent, setprotoent, setservent

Время

gmtime, localtime, time, times

Функции Perl в алфавитном порядке

Многие из последующих имен функций аннотированы значками. Вот их значение:

- [$_] Использует $_ ($ARG) в качестве переменной по умолчанию.
- [$!] Устанавливает $! ($OS_ERROR) при ошибках системных вызовов.
- [$@] Возбуждает исключительные ситуации; используйте eval для перехвата $@ ($EVAL_ERROR).
- [$?] Устанавливает $? ($CHILD_ERROR) при выходе из дочернего процесса.
- [T] Делает возвращаемые данные мечеными.
- [T] Делает возвращаемые данные мечеными при некоторых установках системы, национальных настройках или установках указателей.
- [X ARG] Возбуждает исключительную ситуацию при получении аргумента недопустимого типа.
- [X RO] Возбуждает исключительную ситуацию при модификации объекта, доступного только для чтения.
- [X T] Возбуждает исключительную ситуацию при получении помеченных данных.
- [X U] Возбуждает исключительную ситуацию, если не реализована на данной платформе.

Функции, которые возвращают помеченные данные при получении таких данных на входе, не выделены, поскольку это делает большинство функций. В частности, если любая функция применяется к %ENV или @ARGV, то на выходе получаются помеченные данные.

Функции, помеченные значком [X ARG], возбуждают исключительную ситуацию, когда они требуют, но не получают аргумент определенного типа (например, указатель файла для операций ввода-вывода, ссылки для bless и т. д.).

Функциям, помеченным значком [X RO], иногда требуется изменить свои аргументы. Если они не могут изменить свой аргумент, поскольку для него установлен атрибут «только для чтения», то возбуждают исключительную ситуа-

цию. Примерами переменных «только для чтения» служат специальные переменные, содержащие данные, захваченные при поиске по шаблону, и переменные, являющиеся псевдонимами констант.

Функции со значком [X U] могут быть реализованы не на всех платформах. Хотя многие из них получили свои имена от функций в библиотеке C Unix, не надо думать, что те, кто работает не в Unix, не могут ими пользоваться. Многие из них эмулируются, даже те, для которых мы этого не ожидаем: например, `fork` в системах Win32, которая работает в версии 5.6 Perl. Дополнительную информацию о переносимости и специфических для системы функциях можно найти на странице руководства *perlport*, а также в специфической для платформы документации, поступившей с версией Perl для конкретной платформы.

Функции, которые возбуждают другие различные исключительные ситуации, помечаются значком [$@], в том числе математические функции, возбуждающие ошибки диапазона аргументов, например `sqrt(-1)`.

abs

[$_]

```
abs VALUE
abs
```

Эта функция возвращает абсолютное значение своего аргумента.

```
$diff = abs($first - $second);
```

Замечание: здесь и в последующих примерах хороший стиль программирования (и директива `use strict`) требует добавления модификатора `my` для объявления новой переменной с лексической областью видимости, например:

```
my $diff = abs($first - $second);
```

Однако для ясности мы опускаем `my` в большинстве примеров. Просто считайте, что все такие переменные были объявлены раньше, если это вам нравится.

accept

[$!] [X ARG] [X U]

```
accept SOCKET, PROTOSOCKET
```

Эту функцию используют процессы на сервере, которые собираются ждать подключений клиентов через сокеты. *PROTOSOCKET* должен быть указателем файла, уже открытым оператором `socket` и привязанным к одному из сетевых адресов сервера или к `INADDR_ANY`. Выполнение приостанавливается до осуществления соединения, при котором указатель файла *SOCKET* открывается и прикрепляется к вновь созданному соединению. Исходный *PROTOSOCKET* остается без изменений; его единственное назначение состоит в том, чтобы быть клонированным в реальный сокет. Функция возвращает адрес, с которым произведено соединение, при успешном обращении и «ложь» в противном случае. Например:

```
unless ($peer = accept(SOCK, PROTOSOCK)) {
    die "Невозможно принять соединение: $!\n";
}
```

В системах, которые это разрешают, будет установлен флаг закрытия при exec для вновь открытого дескриптора файла, если этого требует значение $^F ($SYSTEM_FD_MAX).

См. *accept*(2). См. также пример в разделе «Сокеты» главы 16 «Межпроцессное взаимодействие».

alarm

```
alarm EXPR
alarm
```

Эта функция посылает текущему процессу сигнал SIGALRM по истечении *EXPR* секунд.

Одновременно может быть активен только один таймер. Каждое обращение к функции отключает предшествующий таймер. Значение *EXPR*, равное 0, можно использовать для отключения предыдущего таймера без запуска нового. Возвращаемое значение представляет остаток времени предыдущего таймера.

```
print "Ответь мне в течение минуты или умри: ";
alarm(60);              # убить программу через минуту
$answer = <STDIN>;
$timeleft = alarm(0);  # сбросить таймер
print "У вас оставалось $timeleft секунд\n";
```

Смешивание вызовов alarm и sleep обычно является ошибкой, потому что во многих системах для реализации *sleep*(3) используется механизм системного вызова *alarm*(2). На более старых машинах истекшее время может быть на одну секунду меньше заданного из-за способа подсчета секунд. Кроме того, сильно загруженная система может не сразу приступить к выполнению процесса. См. в главе 16 сведения об обработке сигналов.

Чтобы задавать время подачи сигналов с большей точностью, чем одна секунда, можно применить функцию syscall для обращения к *setitimer*(2), если система это поддерживает. Модуль CPAN Time::HiRes также предоставляет функции для этой цели.

atan2

```
atan2 Y, X
```

Эта функция возвращает главное значение арктангенса *Y/X* в диапазоне от $-\pi$ до π. Быстро получить приближенное значение π можно так:

```
$pi = atan2(1,1) * 4;
```

Для вычисления тангенсов можно обратиться к функции tan из модулей Math::Trig или POSIX либо просто использовать известное соотношение:

```
sub tan { sin($_[0]) / cos($_[0]) }
```

bind

$! X ARG X U X T

```
bind SOCKET, NAME
```

Эта функция прикрепляет адрес (имя) к уже открытому сокету, задаваемому уже открытым указателем файла *SOCKET*. Функция возвращает значение «истина» в случае успеха и «ложь» в противном случае. *NAME* должно быть упакованным адресом сокета, имеющим надлежащий тип.

```
use Socket;
$port_number = 80;      # допустим, мы хотим быть веб-сервером
$sockaddr = sockaddr_in($port_number, INADDR_ANY);
bind SOCK, $sockaddr or die "Невозможно связать $port_number: $!\n";
```

См. *bind*(2). См. также примеры в разделе «Сокеты» главы 16.

binmode

X ARG

```
binmode FILEHANDLE, DISCIPLINES
binmode FILEHANDLE
```

Эта функция устанавливает для указателя файла *FILEHANDLE* семантику, заданную аргументом *DISCIPLINES*. Если аргумент *DISCIPLINES* опущен, то к указателю файла применяется двоичная (или raw) семантика. Если *FILEHANDLE* представляет собой выражение, то его значение принимается за имя указателя файла или ссылку на указатель файла в зависимости от обстоятельств.

Функцию binmode нужно вызывать после open, но перед осуществлением любого ввода-вывода по указателю файла. Единственным способом переустановить режим указателя файла является повторное открытие файла, поскольку различные дисциплины могут хранить разные фрагменты данных в разных буферах. Это ограничение в будущем может быть ослаблено.

В прежние времена функция binmode применялась в основном в операционных системах, библиотеки времени исполнения которых различали текстовые и двоичные файлы. В таких системах назначением binmode было отключение текстовой семантики, использовавшейся по умолчанию. Однако с приходом Unicode все программы на всех системах должны каким-то образом замечать различие, даже в системах Unix и Mac. В наше время есть только один тип двоичного файла (если говорить о Perl), но много типов текстовых файлов, которые тоже было бы желательно обрабатывать в Perl одним способом. Поэтому в Perl есть единственный внутренний формат для текста Unicode: UTF-8. Поскольку существует много типов текстовых файлов, то часто требуется при вводе транслировать текстовые файлы в UTF-8, а при выводе в какой-либо из унаследованных наборов символов или в другое представле-

ние Unicode. Посредством этих дисциплин можно сообщить Perl, как точно (или не точно) осуществлять такие переводы.[1]

Например, дисциплина ":text" укажет Perl на необходимость общей обработки текста, не сообщая, какого рода обработка текста должна быть проведена. Но такие дисциплины, как ":utf8" и ":latin1", указывают Perl, в каком текстовом формате производить чтение и запись. С другой стороны, дисциплина ":raw" указывает Perl, чтобы он не трогал данные своими «шаловливыми ручонками». Подробнее о том, как работают (или будут работать) дисциплины, сказано в описании функции open. Оставшаяся часть данного изложения посвящена тому, что делает binmode в отсутствие аргумента *DISCIPLINES*, т. е. историческому значению binmode, которое эквивалентно:

```
binmode FILEHANDLE, ":raw";
```

Если не указано другое, Perl предполагает, что заново открытый файл нужно читать и писать в текстовом режиме. Текстовый режим означает, что внутренним указателем конца строки будет \n (перевод строки). Все системы используют \n в качестве внутреннего указателя конца строки, но его действительное представление различается от системы к системе, устройства к устройству и даже от файла к файлу, в зависимости от способа обращения к файлу. В унаследованных от прошлого системах (включая MS-DOS и VMS) то, что программа видит как \n, может быть не тем, что физически хранится на диске. Операционная система может, например, сохранять текстовые файлы с последовательностями \cM\cJ, которые транслируются при вводе и представляются программе как \n, а \n из программы при выводе в файл обратно транслируются в \cM\cJ. Функция binmode отключает эту автоматическую трансляцию в таких системах.

В отсутствие аргумента *DISCIPLINES* функция binmode не оказывает никакого действия в Unix и Mac OS, в которых \n служит для обозначения конца каждой строки и представляет его одним символом. (Однако это могут быть разные символы: Unix использует \cJ, а старые Mac'и – \cM. Все это не имеет значения.)

В следующем примере показано, как сценарий Perl может прочесть изображение GIF из файла и вывести его на стандартное устройство вывода. В системах, которые переводят буквальные данные в нечто отличное от их точного физического представления, необходимо подготовить оба указателя. Хотя можно непосредственно использовать дисциплину ":raw" при открытии GIF, нельзя так просто сделать это с предварительно открываемыми указателями файлов, такими как STDOUT:

```
binmode STDOUT;
open(GIF, "vim-power.gif") or die "Невозможно открыть vim-power.gif: $!\n";
binmode GIF;
```

[1] Точнее, вы будете иметь возможность использовать для этого дисциплины, но во время написания данной книги их реализация не была закончена.

```
while (read(GIF, $buf, 1024)) {
    print STDOUT $buf;
}
```

bless

```
bless REF, CLASSNAME
bless REF
```

Эта функция сообщает объекту ссылки, на который указывает ссылка *REF*, что с этих пор он является объектом в пакете *CLASSNAME* – или текущем пакете, если имя класса *CLASSNAME* не задано. Если *REF* не является действующей ссылкой, создается исключительная ситуация. Для удобства `bless` возвращает ссылку, т. к. часто это последняя функция в подпрограмме-конструкторе. Например:

```
$pet = Beast->new(TYPE => "cougar", NAME => "Clyde");

# then in Beast.pm:
sub new {
    my $class  = shift;
    my %attrs = @_;
    my $self   = { %attrs };
    return bless($self, $class);
}
```

Как правило, следует «освящать» объекты в классы, имена которых имеют смешанный регистр. Пространства имен с именами на нижнем регистре зарезервированы для внутреннего использования в качестве прагм Perl (директив компилятору). Имена встроенных типов (например, «`SCALAR`», «`ARRAY`», «`HASH`» и т. д., не говоря уже о базовом классе всех классов «`UNIVERSAL`») составлены из символов верхнего регистра, поэтому таких имен тоже желательно избегать.

Проверьте, чтобы *CLASSNAME* не было «ложью»; освящение в ложные пакеты не поддерживается и может привести к непредсказуемым результатам.

Отсутствие соответствующего оператора `curse` (проклясть) ошибкой не является. (Зато есть оператор `sin` (грех).) См. также главу 12 «Объекты», где подробнее рассказано об освящении (и благодеяниях) объектов.

caller

```
caller EXPR
caller
```

Эта функция возвращает сведения о вызовах текущей программы и подобные им. Если аргумента нет, она возвращает имя пакета, имя файла и номер строки, из которой была вызвана выполняемая в данный момент подпрограмма:

```
($package, $filename, $line) = caller;
```

Вот пример исключительно привередливой функции, использующей специальные лексемы __PACKAGE__ и __FILE__, описываемые в главе 2 «Всякая всячина»:

```
sub careful {
    my ($package, $filename) = caller;
    unless ($package eq __PACKAGE__ && $filename eq __FILE__) {
        die "Ты не должен был вызывать меня, $package!\n";
    }
    print "свободно вызвал меня \n";
}

sub safecall {
    careful();
}
```

При вызове с аргументом caller вычисляет *EXPR* как количество кадров стека, на которое надо вернуться назад от текущего. Например, аргумент 0 означает текущий кадр стека, 1 означает вызвавшего, 2 означает того, кто вызвал вызвавшего, и т. д. Функция возвращает также дополнительную информацию, как показано ниже:

```
$i = 0;
while (($package, $filename, $line, $subroutine,
        $hasargs, $wantarray, $evaltext, $is_require,
        $hints, $bitmask) = caller($i++) )
{
    ...
}
```

Если кадр представляет собой вызов подпрограммы, $hasargs является «истиной», если имеется собственный массив @_ (не заимствованный у вызвавшего). В противном случае $subroutine может иметь значение "(eval)", если кадр является не вызовом подпрограммы, а eval. В этом случае устанавливаются дополнительные элементы $evaltext и $is_require: $is_require имеет значение «истина», если кадр создан командой require или use, а $evaltext содержит текст команды eval *EXPR*. В частности, для команды eval *BLOCK* элемент $filename есть "(eval)", а $evaltext не определен. (Обратите также внимание, что каждая команда use создает кадр require внутри кадра eval *EXPR*.) Элементы $hints и $bitmask представляют собой внутренние значения: пожалуйста, не используйте их, если только вы не принадлежите к кругу чудотворцев.

В приступе еще более глубокого волшебства caller также заполняет массив @DB::args аргументами, переданными в данном кадре стека, но только при вызове из пакета DB. См. главу 20 «Отладчик Perl».

chdir

```
chdir EXPR
chdir
```

Эта функция изменяет текущий каталог процесса на *EXPR*, если это возможно. Если *EXPR* опущено, подразумевается исходный каталог вызвавшего процесса. Функция возвращает значение «истина» в случае успеха и «ложь» в противном случае.

```
chdir "$prefix/lib" or die "Невозможно cd в $prefix/lib: $!\n";
```

См. также модуль Cwd, описываемый в главе 32 «Стандартные модули», который позволяет автоматически отслеживать текущий каталог.

chmod

```
chmod LIST
```

Эта функция изменяет права доступа для списка файлов. Первым элементом списка должен быть режим в виде числа, как в системном вызове *chmod*(2). Функция возвращает количество успешно измененных файлов. Например,

```
$cnt = chmod 0755, 'file1', 'file2';
```

установит $cnt в 0, 1 или 2 в зависимости от того, сколько файлов было изменено. Успех определяется отсутствием ошибки, а не фактическим изменением, поскольку файл мог иметь тот же режим до выполнения операции. Ошибка может означать, что у пользователя недостаточно прав на изменение режима, т. к. он не является владельцем файла или суперпользователем. Проверьте $!, чтобы найти фактическую причину неудачи.

Вот более типичное применение:

```
chmod(0755, @executables) == @executables
        or die "невозможно chmod некоторых @executables: $!";
```

Для того чтобы узнать, какие файлы не позволили осуществить изменение, сделайте что-либо такое:

```
@cannot = grep {not chmod 0755, $_} 'file1', 'file2', 'file3';
die "$0: could not chmod @cannot\n" if @cannot;
```

В этой идиоме используется функция grep для отбора только тех элементов списка, применение chmod к которым оказалось неудачным.

При использовании нелитеральных данных режима может понадобиться преобразовать восьмеричную строку в число с помощью функции oct. Дело в том, что Perl не принимает автоматически строку за восьмеричное число только потому, что она начинается с «0».

```
$DEF_MODE = 0644;   # Здесь нельзя использовать кавычки!
PROMPT: {
    print "New mode? ";
    $strmode = <STDIN>;
        exit unless defined $strmode;   # проверка eof
    if ($strmode =~ /^\s*$/) {          # проверка пустой строки
```

```
        $mode = $DEF_MODE;
    }
    elsif ($strmode !~ /^\d+$/) {
        print "Требуется числовой режим, а не $strmode\n";
        redo PROMPT;
    }
    else {
        $mode = oct($strmode);          # преобразует "755" в 0755
    }
    chmod $mode, @files;
}
```

Эта функция работает с числовыми режимами во многом сходно с системным вызовом Unix *chmod*(2). Для того чтобы воспользоваться символическим интерфейсом, подобным предоставляемому командой *chmod*(1), обратитесь к модулю `File::chmod` в CPAN.

Можно также импортировать константы `S_I*` из модуля `Fcntl`:

```
use Fcntl ':mode';
chmod S_IRWXU|S_IRGRP|S_IXGRP|S_IROTH|S_IXOTH, @executables;
```

Некоторые считают, что это читается легче, чем `0755`. Решайте сами.

chomp

```
chomp VARIABLE
chomp LIST
chomp
```

Эта функция (обычно) удаляет замыкающий символ перевода строки, находящейся в строке, которая содержится в переменной. Это несколько более безопасная версия `chop` (описываемая следующей), поскольку она не оказывает воздействия на строку, которая не оканчивается символом перевода строки. Точнее, она удаляет строку-терминатор в соответствии с текущим значением `$/`, а не просто любой последний символ.

В отличие от `chop`, функция `chomp` возвращает число удаленных символов. Если значением `$/` является `""` (режим параграфов, или абзацев), `chomp` удаляет все замыкающие символы перевода строки из указанной строки (или строк, если аргументом является *LIST*). Нельзя применять `chomp` к литералам, можно только к переменным.

Например:

```
while (<PASSWD>) {
    chomp;   # игнорировать \n в последнем поле
    @array = split /:/;
    ...
}
```

В версии 5.6 значение `chomp` немного изменилось, поскольку дисциплинам ввода разрешено обходить значение переменной `$/` и размечать точки усече-

ния строк. Преимущество в том, что дисциплина ввода может распознавать несколько видов окончаний строк (например, разделители абзацев и строк Unicode), но, тем не менее, chomp обеспечивает безопасное удаление окончания текущей строки.

chop

```
chop VARIABLE
chop LIST
chop
```

Эта функция «обрубает» последний символ строковой переменной и возвращает его. Оператор chop применяется в основном для удаления символа перевода строки в конце вводимой записи, что более эффективно, чем использование подстановки. Если это все, что вам нужно, то безопаснее применять chomp, поскольку chop всегда укорачивает строку независимо от того, что в ней находилось, кроме того, chomp более разборчива.

Нельзя применять chop к литералу, можно только к переменной.

Если chop применяется к списку переменных, то усекается каждая строка списка:

```
@lines = `cat myfile`;
chop @lines;
```

Применять chop можно ко всему, что является левым значением, в том числе к присвоению:

```
chop($cwd = `pwd`);
chop($answer = <STDIN>);
```

Это – не то же, что

```
$answer = chop($tmp = <STDIN>);  # НЕВЕРНО
```

приводящее к помещению перевода строки в $answer, потому что chop возвращает «отрубленный» символ, а не оставшуюся строку (которая находится в $tmp). Один из способов получить предполагавшийся результат состоит в применении substr:

```
$answer = substr <STDIN>, 0, -1;
```

Но чаще это записывается так:

```
chop($answer = <STDIN>);
```

В самом общем случае chop можно выразить через substr:

```
$last_char = chop($var);
$last_char = substr($var, -1, 1, "");   # то же самое
```

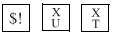
Поняв эту эквивалентность, можно удалять больше одного символа. Для этого следует использовать substr в качестве левого значения, присваивая ему нулевую строку. Следующий код удаляет последние пять символов $caravan:

```
substr($caravan, -5) = "";
```

Отрицательный индекс заставляет substr вести отсчет от конца строки, а не от начала. Сохранить удаляемые таким способом символы можно при помощи формата substr с четырьмя аргументами, создав некий пятикратный вариант chop:

```
$tail = substr($caravan, -5, 5, "");
```

chown



```
chown LIST
```

Эта функция изменяет владельца и группу для списка файлов. Первые два элемента списка должны быть *числовыми* UID и GID в указанном порядке. Значение -1 в любой позиции интерпретируется в большинстве систем как необходимость оставить элемент без изменений. Функция возвращает число успешно модифицированных файлов. Например:

```
chown($uidnum, $gidnum, 'file1', 'file2') == 2
        or die "невозможно chown file1 или file2: $!";
```

установит $cnt в 0, 1 или 2 в зависимости от того, сколько файлов было изменено (в смысле успеха операции, а не в смысле того, что владелец изменился). Вот более типичное применение:

```
chown($uidnum, $gidnum, @filenames) == @filenames
        or die "невозможно chown @filenames: $!";
```

Ниже приведена подпрограмма, которая принимает имя пользователя, ищет ID пользователя и группы и выполняет chown:

```
sub chown_by_name {
    my($user, @files) = @_;
    chown((getpwnam($user))[2,3], @files) == @files
            or die "невозможно chown @files: $!";
}

chown_by_name("fred", glob("*.c"));
```

Однако может и не требоваться, чтобы группа изменялась, как это делает предыдущая функция, потому что файл */etc/passwd* ассоциирует каждого пользователя с единственной группой, даже несмотря на то, что пользователь может быть членом многих вторичных групп в соответствии с */etc/group*. Выход в том, чтобы передать -1 в качестве GID, в результате чего группа не меняется. Если передать -1 в качестве UID и допустимый GID, то можно установить группу, не меняя владельца.

В большинстве систем менять владельца файла разрешается только суперпользователю, хотя у вас должна быть возможность изменить группу на любую из своих вторичных групп. В недостаточно защищенных системах эти ограничения могут быть ослаблены, но такое допущение не переносимо. В POSIX-системах можно определить, какое правило используется, следующим образом:

```
use POSIX qw(sysconf _PC_CHOWN_RESTRICTED);
# пытаться выполнить, если мы суперпользователь или в слабой системе
if ($> == 0 || !sysconf(_PC_CHOWN_RESTRICTED) ) {
    chown($uidnum, -1, $filename)
        or die "невозможно chown $filename на $uidnum: $!";
}
```

chr

$_

```
chr NUMBER
chr
```

Эта функция возвращает символ, представленный в наборе символов числом *NUMBER*. Например, `chr(65)` соответствует символу «`A`» и в ASCII, и в Unicode, а `chr(0x263a)` является смайликом в Unicode. В качестве функции, обратной для `chr`, используется `ord`.

Если вы предпочитаете задавать символы именами, а не числами (например, `"\N{WHITE SMILING FACE}"` для смайлика Unicode), ознакомьтесь с описанием директивы `charnames` в главе 31 «Модули прагм».

chroot

$_ $! X U X T

```
chroot FILENAME
chroot
```

При успешном выполнении *FILENAME* становится новым корневым каталогом для текущего процесса – отправной точкой маршрутов, начинающихся с «/». Этот каталог наследуется в вызовах `exec` и всеми подпроцессами, ответвляемыми `fork` после вызова `chroot`. Способа отменить `chroot` нет. По соображениям безопасности эту функцию может вызывать только суперпользователь. Вот код, который делает примерно то же, что делают многие серверы FTP:

```
chroot((getpwnam('ftp'))[7])
    or die "Не разрешен анонимный ftp: $!\n";
```

Эта функция, скорее всего, не будет работать на системах, отличных от Unix. См. *chroot*(2).

close

$! $? X ARG

```
close FILEHANDLE
close
```

Эта функция закрывает файл, сокет или канал, связанные с указателем *FILEHANDLE*. (Она закрывает текущий выбранный указатель, если аргумент опущен.) Она возвращает значение «истина», если закрытие успешно, и «ложь» в противном случае. Не требуется закрывать *FILEHANDLE*, если сразу за этим предполагается выполнить для него новый вызов open, поскольку очередной open сам закроет его. (См. open.) Однако явное применение close к входному файлу влечет сброс счетчика строк ($.), тогда как неявное открытие с помощью open – нет.

FILEHANDLE может представляь собой выражение, значение которого используется в качестве косвенного указателя файла (либо реального имени указателя файла, либо ссылки на то, что может быть интерпретировано как объект указателя файла).

Если указатель файла получен в результате открытия канала, то close возвращает значение «ложь» при неудаче какого-либо из выполняющихся в нем системных вызовов или если программа на другом конце канала завершила работу с ненулевым статусом. В последнем случае close устанавливает $! ($OS_ERROR) в ноль. Поэтому, если close для канала возвращает ненулевой статус, проверьте $!, чтобы выяснить, была ли проблема связана с самим каналом (ненулевое значение) или с программой на другом конце (нулевое значение). В любом случае $? ($CHILD_ERROR) содержит значение статуса ожидания (см. его интерпретацию в описании функции system) команды, связанной с другим концом канала. Например:

```
open(OUTPUT, '| sort -rn | lpr -p') # канал для sort и lpr
    or die "Невозможно запустить канал sortlpr: $!";
print OUTPUT @lines;                # вывести что-то в output
close OUTPUT                        # ждать завершения sort
    or warn $! ? "Системная ошибка при закрытии канала sortlpr: $!"
               : "Статус ожидания $? от канала sortlpr";
```

Указатель файла, получаемый в результате применения *dup*(2) к каналу, рассматривается как обычный указатель файла, поэтому close не будет ждать дочернего процесса по этому указателю. Ожидание дочернего процесса следует реализовывать посредством закрытия исходного указателя файла. Например:

```
open(NETSTAT, "netstat -rn |")
    or die "can't run netstat: $!";
open(STDIN, "<&NETSTAT")
    or die "can't dup to stdin: $!";
```

Если закрыть здесь STDIN, то ожидания нет, а если NETSTAT, то есть.

Если каким-то образом ухитриться убрать завершившийся дочерний процесс канала самостоятельно, то закрытие окажется неудачным. Это может произойти, если у вас есть собственный обработчик $SIG{CHLD}, который запустился при выходе из дочернего процесса канала, или вы намеренно вызвали waitpid с ID процесса, возвращенным при вызове open.

closedir

$! X ARG X U

```
closedir DIRHANDLE
```

Эта функция закрывает каталог, открытый с помощью opendir, и возвращает значение, соответствующее успешности этой операции. См. примеры в описании readdir. Параметр *DIRHANDLE* может быть выражением, значение которого используется как косвенный указатель каталога, обычно являющийся реальным именем указателя каталога.

connect

$! X ARG X T X U

```
connect SOCKET, NAME
```

Эта функция инициирует соединение с другим процессом, который находится в ожидании после accept. Функция возвращает значение «истина» в случае успеха и «ложь» в противном случае. *NAME* должно быть упакованным сетевым адресом сокета надлежащего типа. Приведем пример, предполагающий, что SOCK является ранее созданным сокетом:

```
use Socket;

my ($remote, $port) = ("www.perl.com", 80);
my $destaddr = sockaddr_in($port, inet_aton($remote));
connect SOCK, $destaddr
    or die "Can't connect to $remote at port $port: $!";
```

Чтобы отсоединить сокет, используйте close или shutdown. См. также примеры в разделе «Сокеты» главы 16. См. также *connect*(2).

cos

$_

```
cos EXPR
cos
```

Эта функция возвращает косинус *EXPR* (выраженный в радианах). Например, следующий сценарий выведет таблицу косинусов углов, измеренных в градусах:

```
# "Ленивый" способ перевода градусов в радианы.

$pi = atan2(1,1) * 4;
$piover180 = $pi/180;

# Вывод таблицы.
for ($deg = 0; $deg <= 90; $deg++) {
    printf "%3d %7.5f\n", $deg, cos($deg * $piover180);
}
```

Операция, обратная взятию косинуса, выполняется посредством функции acos() из модулей Math::Trig или POSIX или при помощи соотношения:

```
sub acos { atan2( sqrt(1 - $_[0] * $_[0]), $_[0] ) }
```

crypt

crypt *PLAINTEXT*, *SALT*

Эта функция вычисляет однонаправленный хеш строки точно в стиле *crypt*(3). Это до некоторой степени полезно для проверки файла паролей на наличие плохих паролей,[1] хотя на самом деле задача в первую очередь заключается в том, чтобы не допустить добавления плохих паролей.

Функция crypt задумана как однонаправленная, и это можно сравнить с разбиванием яиц для приготовления омлета. Не существует (известного) способа расшифровать зашифрованный пароль, кроме исчерпывающего перебора с помощью грубой силы.

При проверке зашифрованной строки нужно использовать зашифрованный текст в качестве *SALT* (например, crypt($plain, $crypted) eq $crypted). Это позволяет программе работать как со стандартным crypt, так и с более экзотическими реализациями.

Выбирая заново *SALT*, необходимо, как минимум, создать случайную строку из двух символов, принадлежащих множеству [./0-9A-Za-z] (например, путем join '', ('.', '/', 0..9, 'A'..'Z', 'a'..'z')[rand 64, rand 64]). В старых реализациях crypt требовались только первые два символа из *SALT*, но код, дающий только два первые символа, считается теперь непереносимым. Интересные подробности можно найти на странице руководства для *crypt*(3).

Вот пример, в котором проверяется, знает ли тот, кто работает с этой программой, свой пароль:

```
$pwd = (getpwuid ($<))[1];    # Предполагается, что мы в Unix.

system "stty -echo";   # или см. Term::ReadKey в CPAN
print "Password: ";
chomp($word = <STDIN>);
print "\n";
system "stty echo";

if (crypt($word, $pwd) ne $pwd) {
    die "Sorry...\n";
} else {
    print "ok\n";
}
```

Разумеется, вводить свой пароль по просьбе всякого, кто об этом попросит, неблагоразумно.

Файлы теневых паролей несколько более безопасны, чем традиционные файлы паролей, и для доступа к ним могут потребоваться права суперпользователя. Поскольку немногие программы должны выполняться с такими мощными правами, можно сделать так, чтобы программа поддерживала соб-

[1] Это разрешается делать только тем, кто имеет достойные намерения.

ственную независимую систему аутентификации, сохраняя обработанные crypt строки в файлах, отличных от */etc/passwd* и */etc/shadow*.

Функция crypt не пригодна для зашифровывания больших объемов данных хотя бы потому, что вы не сможете получить свою информацию обратно. Загляните в каталоги *by-module/Crypt* и *by-module/PGP* на своем любимом зеркале CPAN, и вы найдете множество модулей, которые могут оказаться полезными.

dbmclose

```
dbmclose HASH
```

Эта функция разрывает связь между файлом DBM (базы данных) и хешем. dbmclose в действительности является просто вызовом untie с надлежащими аргументами, но оставлена для совместимости со старыми версиями Perl.

dbmopen

```
dbmopen HASH, DBNAME, MODE
```

Эта функция привязывает файл DBM к хешу (т. е. ассоциативному массиву). (DBM означает «database management» – управление базой данных – и состоит из набора программ библиотеки C, обеспечивающих произвольный доступ к записям с использованием алгоритма хеширования.) *HASH* является именем хеша (вместе с %). *DBNAME* представляет собой имя базы данных (без каких-либо расширений типа *.dir* или *.pag*). Если база данных не существует и задан допустимый режим *MODE*, то база данных создается; защита определяется аргументом *MODE*, модифицированным функцией umask. Чтобы предотвратить создание новой базы данных, можно задать в *MODE* значение undef, и функция возвратит «ложь», если не сможет найти указанную базу данных. Значения, присвоенные хешу до вызова dbmopen, являются недоступными.

Функция dbmopen в действительности представляет собой просто вызов tie с надлежащими аргументами, но оставлена для совместимости с «древними» версиями Perl. Управлять тем, какая библиотека DBM используется, можно непосредственно с помощью интерфейса tie либо путем загрузки соответствующего модуля перед вызовом dbmopen. Вот пример, который работает на некоторых системах с версиями DB_File, сходными с версиями в броузере Netscape:

```
use DB_File;
dbmopen(%NS_Hist, "$ENV{HOME}/.netscape/history.dat", undef)
    or die "Невозможно открыть файл netscape history: $!";

while (($url, $when) = each %NS_Hist) {
    next unless defined($when);
    chop ($url, $when);        # удалить замыкающие нулевые байты
    printf "Visited %s at %s.\n", $url,
        scalar(localtime(unpack("V",$when)));
}
```

Если у пользователя нет права записи в файл DBM, то он может только читать переменную хеша, но не записывать в нее. Узнать, есть ли у вас право записи, можно посредством проверки файла вида `-w $file`, можно также попытаться установить ненастоящий элемент хеша внутри `eval {}`, которая перехватит исключительную ситуацию.

Такие функции, как `keys` и `values`, могут возвращать очень большие списки значений при работе с большими файлами DBM. Более предпочтительным для прохода по большим файлам DBM может оказаться применение функции `each`, что позволяет не загружать в память сразу весь файл.

Для хешей, связанных с файлами DBM, действуют такие же ограничения, как в используемом типе пакета DBM, в том числе ограничение объема размещаемых данных. В случае работы с короткими ключами и значениями такие проблемы возникают редко. См. также описание модуля `DB_File` в главе 32 «Стандартные модули».

Следует также помнить, что многие существующие базы данных DBM содержат ключи и значения, заканчивающиеся символом `null`, поскольку создавались для программ на C. Примерами являются файл журнала Netscape и старый файл псевдонимов *sendmail*. Нужно просто использовать `"$key\0"` для извлечения значения, а затем удалять из него `null`:

```
$alias = $aliases{"postmaster\0"};
chop $alias;   # удалить null
```

В настоящее время нет встроенного средства блокировки общего DBM-файла. Некоторые считают это ошибкой. Модуль `GDBM_File` делает попытку обеспечить блокировку на уровне файла в целом. При наличии сомнений лучше прибегнуть к отдельной блокировке файла.

defined

```
defined EXPR
defined
```

Эта функция возвращает булево значение, говорящее о том, имеет *EXPR* определенное значение или нет. По большей части данные, с которыми вы имеете дело, являются определенными, но скаляр, который не содержит допустимого строкового, числового или ссылочного значения, считается содержащим неопределенное значение, или, для краткости, `undef`. Инициализация скалярной переменной конкретным значением делает ее определенной, и такой она остается, пока ей не будет явно присвоено неопределенное значение или для нее не будет вызвана функция `undef`.

Многие операции возвращают `undef` при возникновении исключительной ситуации, например, при достижении конца файла, при использовании значения неопределенной переменной, ошибке операционной системы и т. д. Поскольку `undef` является разновидностью значения «ложь», простая булева проверка не различает `undef`, числовой ноль, нулевую строку и строку из од-

ного символа «0» – все они являются одинаково ложными. Функция defined позволяет отличить неопределенную нулевую строку от определенной нулевой строки, когда применяются операторы, которые действительно могут возвращать нулевую строку.

Вот фрагмент, который проверяет скалярное значение из хеша:

```
print if defined $switch{D};
```

В случае такого применения с элементом хеша defined только сообщает, является ли значение определенным, а не то, есть ли такой ключ в хеше. Допустимо иметь ключ, значение которого является неопределенным, тем не менее, сам ключ существует. Используйте exists, чтобы определить, существует ли ключ.

В следующем примере действует соглашение, согласно которому некоторые операции возвращают неопределенное значение, когда кончаются данные:

```
print "$val\n" while defined($val = pop(@ary));
```

А в следующем примере мы делаем то же самое с функцией getpwent, чтобы извлечь информацию о пользователях системы.

```
setpwent();
while (defined($name = getpwent())) {
    print "<<$name>>\n";
}
endpwent();
```

То же можно использовать с ошибками, возвращаемыми системными вызовами, для которых допустимо возвращать значение «ложь»:

```
die "Can't readlink $sym: $!"
    unless defined($value = readlink $sym);
```

Можно применять defined и для того, чтобы проверить, определена ли подпрограмма. Это позволяет избежать аварии при вызове несуществующих подпрограмм (или подпрограмм, которые объявлены, но не имеют определения):

```
indir("funcname", @arglist);
sub indir {
    my $subname = shift;
    no strict 'refs';  # чтобы использовать subname косвенно
    if (defined &$subname) {
        &$subname(@_);    # или $subname->(@_);
    }
    else {
        warn "Игнорирование вызова недопустимой функции $subname";
    }
}
```

Использование defined с составными объектами (хешами и массивами) является устаревшим. (Оно сообщало о том, выделялась ли память под этот

объект.) Вместо этого рекомендуем простую булеву проверку числа элементов массива или хеша:

```
if (@an_array) { print "есть в массиве элементы\n" }
if (%a_hash)   { print "есть в хеше элементы\n"   }
```

См. также undef и exists.

delete

```
delete EXPR
```

Эта функция удаляет элемент (или срез элементов) из указанного хеша или массива. (См. unlink, если требуется удалить файл.) Удаленные элементы возвращаются в указанном порядке, хотя для связанных переменных, например файлов DBM, это не гарантируется. После операции удаления функция exists возвращает «ложь» для любого удаленного ключа или индекса. (Для сравнения: после функции undef функция exists продолжает возвращать «истину», поскольку функция undef только делает значение элемента неопределенным, но не удаляет его.)

Удаление из хеша %ENV модифицирует окружение. Удаление из хеша, связанного с файлом DBM (доступным для записи), удаляет запись из этого файла DBM.

Исторически удаление было возможно только из хешей, но Perl версии 5.6 допускает также удаление из массивов. Удаление из массива ведет к тому, что элемент в указанной позиции возвращается в полностью неинициализированное состояние, но образующийся разрыв при этом не закрывается, т. к. при этом изменились бы позиции последующих элементов. Для этого используйте функцию splice. (Однако если удалить последний элемент массива, массив сократится на единицу (или более, в зависимости от положения нового последнего элемента (если он есть))).

EXPR может иметь произвольную сложность при условии, что последняя операция является поиском в хеше или массиве:

```
# создать массив массивов из хеша
$dungeon[$x][$y] = \%properties;

# удалить одно property из хеша
delete $dungeon[$x][$y]{"OCCUPIED"};

# удалить сразу три properties из хеша
delete @{ $dungeon[$x][$y] }{ "OCCUPIED", "DAMP", "LIGHTED" };

# удалить ссылку на %properties из массива
delete $dungeon[$x][$y];
```

В следующем наивном примере происходит неэффективное удаление всех значений из %hash:

```
foreach $key (keys %hash) {
    delete $hash{$key};
}
```

Так же как и в этом:

```
delete @hash{keys %hash};
```

В обоих случаях результат достигается медленнее, чем в случае присвоения пустого списка (если сделать хеш неопределенным, тоже получится быстрее):

```
%hash = ();          # полностью очистить %hash
undef %hash;         # забыть, что %hash когда-либо существовал
```

Аналогично, для массивов:

```
foreach $index (0 .. $#array) {
    delete $array[$index];
}
```

и:

```
delete @array[0 .. $#array];
```

менее эффективны, чем каждый из:

```
@array = ();          # полностью очистить @array
undef @array;         # забыть, что @array когда-либо существовал
```

die

$@

```
die LIST
die
```

Вне eval эта функция выводит значение, конкатенируемое из списка *LIST*, на STDERR и осуществляет выход с текущим значением $! (переменная errno C-библиотеки). Если $! равна 0, то выход производится со значением $? >> 8 (что является значением статуса последнего убранного дочернего процесса из system, wait, close для канала или `command`). Если $? >> 8 равна 0, то осуществляется выход со значением 255.

Внутри eval эта функция записывает в переменную $@ сообщение об ошибке, которая была бы иначе порождена, а затем выходит из eval, которая возвращает undef. Функция die может, таким образом, применяться для создания именованных исключительных ситуаций, которые могут перехватываться на более высоком уровне программы. См. описание функции eval далее в этой главе.

Если *LIST* — единственная ссылка на объект, то предполагается, что это объект исключительной ситуации, и он возвращается в немодифицированном виде как исключительная ситуация в $@.

Если *LIST* пуст и $@ уже содержит строковое значение (обычно от предшествующей eval), то это значение повторно используется после добавления "\t...propagated". Это полезно для распространения (повторного возбуждения) исключительных ситуаций:

```
eval { ... };
die unless $@ =~ /Ожидаемая исключительная ситуация/;
```

Если *LIST* пуст и $@ уже содержит объект исключительной ситуации, то вызывается метод $@->PROPAGATE, чтобы определить, как должна распространяться ошибка.

Если *LIST* пуст и $@ пуста, то используется строка "Died".

Если последнее значение *LIST* не оканчивается символом перевода строки (и вы не передаете объект исключительной ситуации), то к сообщению дописываются имя файла текущего сценария, номер строки и номер вводимой строки (если такая есть), а также символ перевода строки. Совет: иногда можно сделать сообщение более осмысленным, дописав к нему ", stopped", если добавить строку "at scriptname line 123". Допустим, что выполняется сценарий *canasta*; посмотрите на разницу между двумя способами использования die:

```
die "/usr/games is no good";
die "/usr/games is no good, stopped";
```

которые порождают, соответственно:

```
/usr/games is no good at canasta line 123.
/usr/games is no good, stopped at canasta line 123.
```

Для того чтобы использовать собственные сообщения об ошибках, выводящие имя файла и номер строки, используйте специальные лексемы __FILE__ и __LINE__:

```
die '"', __FILE__, '", line ', __LINE__, ", phooey on you!\n";
```

В результате получается такой вывод:

```
"canasta", line 38, phooey on you!
```

Еще одно замечание по поводу стиля: посмотрите на следующие эквивалентные примеры:

```
die "Can't cd to spool: $!\n"   unless chdir '/usr/spool/news';

chdir '/usr/spool/news'         or die "Can't cd to spool: $!\n"
```

Поскольку важной частью является chdir, то вторая форма в целом предпочтительнее.

См. также exit, warn, %SIG и модуль Carp.

do (block)

```
do BLOCK
```

Форма do *BLOCK* выполняет последовательность команд в блоке и возвращает значение последнего выражения, вычисленного в блоке. Если есть модификатор while или until, то Perl один раз выполняет *BLOCK*, прежде чем проверить условие цикла. (В других командах модификаторы цикла сначала проверяют условие.) Сама конструкция do *BLOCK* циклом *не* считается, поэтому команды управления циклом next, last и redo нельзя применять для выхода из блока или его перезапуска. О том, как это обойти, см. в разделе «Голые блоки» главы 4 «Команды и объявления».

do (file)

```
do FILE
```

Форма do *FILE* использует значение *FILE* как имя файла и выполняет содержимое файла как сценарий Perl. Ее основным назначением является (или, скорее, являлось) включение подпрограмм из библиотеки подпрограмм Perl, поэтому:

```
do `stat.pl`;
```

имеет большое сходство с:

```
scalar eval `cat stat.pl`;   # `type stat.pl` в Windows
```

за исключением того, что do более эффективна, более сжата, использует имя текущего файла для сообщений об ошибках, выполняет поиск в каталогах, перечисленных в массиве @INC и обновляет %INC, если файл найден. (См. главу 28 «Специальные имена».) Разница также в том, что код, выполняемый с помощью do *FILE*, не может видеть лексические переменные из охватывающей лексической области видимости, тогда как код в eval *FILE* может. Они одинаковы, однако, в том, что заново производится анализ файла при каждом его вызове; поэтому нежелательно делать это в цикле, если только само имя файла не меняется при каждом проходе цикла.

Если do не может прочесть файл, возвращается undef и $! присваивается ошибка. Если do может прочесть файл, но не может его скомпилировать, возвращается undef, а сообщение об ошибке записывается в $@. Если файл успешно скомпилирован, do возвращает значение последнего вычисленного выражения.

Включение библиотечных модулей (имеющих обязательный суффикс *.pm*) лучше выполнять с помощью операторов use и require, которые также проверяют ошибки и создают исключительную ситуацию при возникновении ошибки. Они обеспечивают и другие преимущества: позволяют избежать повторной загрузки, содействуют объектно-ориентированному программированию и дают советы компилятору по прототипам функций.

Но do *FILE* сохраняет полезность, например, при чтении файлов конфигурации программы. Ручную проверку ошибок можно произвести так:

```
# считать файлы конфигурации: сначала системные, затем пользовательские
for $file ("/usr/share/proggie/defaults.rc",
           "$ENV{HOME}/.someprogrc")
{
    unless ($return = do $file) {
        warn "couldn't parse $file: $@" if $@;
        warn "couldn't do $file: $!"    unless defined $return;
        warn "couldn't run $file"       unless $return;
    }
}
```

Долго работающий демон может периодически проверять временную метку на своем файле конфигурации, и если файл изменился после предыдущего чтения, то демон может использовать do, чтобы перезагрузить этот файл. С помощью do это реализуется более аккуратно, чем с помощью require или use.

do (subroutine)

$@

```
do SUBROUTINE (LIST)
```

do *SUBROUTINE* (*LIST*) представляет собой устаревшую форму вызова подпрограммы. Если *SUBROUTINE* не определена, вызывается исключительная ситуация. См. главу 6 «Подпрограммы».

dump

```
dump LABEL
dump
```

Эта функция вызывает немедленный дамп памяти. В основном это нужно для того, чтобы с помощью программы *undump* (не поставляется) превратить дамп памяти в исполняемый двоичный файл после инициализации своих переменных в начале программы. При выполнении нового двоичного файла сначала выполняется goto *LABEL* (со всеми ограничениями, присущими goto). Считайте, что это goto, в который вмешались дамп памяти и перевоплощение. Если *LABEL* опущена, программа перезапускается с начала. Предупреждение: все файлы, открытые в момент дампа, больше не будут открытыми при перевоплощении программы, что может запутать Perl. См. также описание параметра командной строки -u в главе 19 «Интерфейс командной строки».

Эта функция сейчас значительно устарела, потому что крайне трудно в общем случае преобразовать дамп памяти в исполняемый файл, а также потому, что ее заменили различные серверы компиляции для генерации переносимого байт-кода и компилируемого C-кода.

Тот, кто собирается использовать dump для увеличения скорости выполнения программы, может обратиться к обсуждению проблем эффективности в гла-

ве 24 «Часто используемые приемы программирования», а также генератора родного кода Perl в главе 18 «Компиляция». Можно также рассмотреть возможность автоматической загрузки и самозагрузки, благодаря которым, по крайней мере, *покажется*, что программа выполняется быстрее.

each

```
each HASH
```

Эта функция проходит хеш, выбирая каждый раз одну пару ключ/значение. При вызове в списковом контексте each возвращает двухэлементный список, состоящий из ключа и значения для очередного элемента хеша, что позволяет обойти хеш. При вызове в скалярном контексте each возвращает просто ключ очередного элемента в хеше. Когда хеш прочтен полностью, возвращается пустой список, который при присвоении создает значение «ложь» в скалярном контексте, например при проверке условия цикла. После этого следующее обращение к each снова начнет обход. Вот пример типичного применения, в котором используется предопределенный хеш %ENV:

```
while (($key,$value) = each %ENV) {
    print "$key=$value\n";
}
```

Внутри хеша записи расположены в порядке, кажущемся случайным. Функция each проходит через эту последовательность, потому что каждый хеш помнит, какая запись была возвращена последней. Фактический порядок этой последовательности может измениться в будущих версиях Perl, но гарантируется, что это будет тот же порядок, который функция keys (или values) произведет с тем же (немодифицированным) хешем.

Для каждого хеша существует единственный итератор, совместно используемый всеми вызовами в программе функций each, keys и values; его можно сбросить, прочтя все элементы хеша или вычислив keys %hash или values %hash. Если элементы добавляются или удаляются во время итерации по нему, результирующее поведение становится не вполне определенным: записи могут быть пропущены или повторены.

См. также keys, values и sort.

eof

```
eof FILEHANDLE
eof()
eof
```

Эта функция возвращает значение «истина», если очередное чтение из указателя файла *FILEHANDLE* возвратило бы конец файла или *FILEHANDLE* не открыт. *FILEHANDLE* может быть выражением, значение которого задает реальный указатель файла, либо ссылкой на некоторый объект типа указателя файла. Функция eof без аргументов возвращает статус конца файла для пос-

леднего чтения из файла. Функция eof() с пустыми круглыми скобками () проверяет указатель файла ARGV (который чаще всего виден как нулевой указатель файла в <>). Поэтому внутри цикла while (<>) функция eof() со скобками обнаружит конец только последнего файла группы. Используйте eof (без скобок) для проверки *каждого* файла в цикле while (<>). Например, следующий код вставляет черточки перед последней строкой *последнего* файла:

```
while (<>) {
    if (eof()) {
        print "-" x 30, "\n";
    }
    print;
}
```

Напротив, следующий сценарий сбрасывает нумерацию строк для *каждого* входного файла:

```
# сбрасывать нумерацию строк для каждого входного файла
while (<>) {
    next if /^\s*#/;        # пропустить комментарии
    print "$.\t$_";
} continue {
    close ARGV if eof;      # Не eof()!
}
```

Подобно "$" в программе *sed*, функция eof может появляться в диапазонах номеров строк. Вот сценарий, который выводит строки от /pattern/ до конца каждого входного файла:

```
while (<>) {
    print if /pattern/ .. eof;
}
```

Здесь триггерный оператор (..) вычисляет значение поиска по шаблону для каждой строки. Пока шаблон не будет найден, оператор возвращает «ложь». Когда, наконец, поиск окажется успешным, оператор начинает возвращать «истину», что вызывает печать строк. Когда, наконец, оператор eof возвратит истинное значение (в конце исследуемого файла), оператор триггера сбрасывается и снова начинает возвращать «ложь» для следующего файла в @ARGV.

Предупреждение: функция eof считывает байт и затем возвращает его во входной поток с помощью *ungetc*(3), поэтому в интерактивном контексте от нее нет пользы. На практике опытные программисты Perl редко применяют eof, поскольку различные операторы ввода и так ведут себя прилично в условиях циклов while. См. пример в описании foreach в главе 4 «Команды и объявления».

eval

```
eval BLOCK
eval EXPR
eval
```

Ключевое слово eval служит в Perl двум разным, но связанным целям. Эти цели представлены двумя формами синтаксиса: eval *BLOCK* и eval *EXPR*. Первая форма перехватывает исключительные ситуации (ошибки) этапа исполнения, которые могли бы оказаться фатальными, аналогично блоку «try» в C++ и Java. Вторая форма компилирует и выполняет небольшие фрагменты кода на лету на этапе исполнения, а также перехватывает все исключительные ситуации так же, как первая форма. Однако вторая форма выполняется значительно медленнее первой, поскольку каждый раз строка должна подвергаться синтаксическому анализу. С другой стороны, она является более общей. Какую бы форму вы ни использовали, eval является предпочтительным способом проведения всей обработки исключительных ситуаций в Perl.

Для любой формы eval функция возвращает значение последнего вычисленного выражения так же, как в случае подпрограмм. Аналогично можно использовать оператор return для возврата значения в середине eval. Выражение, предоставляющее возвращаемое значение, вычисляется в пустом, скалярном или списковом контексте в зависимости от контекста самой eval. Дополнительные сведения о том, как выяснить контекст вычисления, можно найти в описании wantarray.

Если возникает перехватываемая ошибка (в том числе полученная при выполнении функции die), eval возвращает undef и помещает сообщение об ошибке (или объект) в $@. Если ошибки нет, $@ устанавливается равной нулевой строке, поэтому можно надежно проверить ее впоследствии на наличие ошибок. Достаточно простой булевой проверки:

```
eval { ... };    # перехват ошибок этапа исполнения
if ($@) { ... }    # обработать ошибку
```

Форма eval *BLOCK* синтаксически проверяется на этапе компиляции, чего вполне достаточно. (Те, кто знаком с медленной формой eval *EXPR*, иногда бывают смущены этим обстоятельством.) Поскольку код в блоке компилируется в то же время, что и код, находящийся вокруг, эта форма eval не может перехватывать исключительные ситуации.

Форма eval *EXPR* может перехватывать синтаксические ошибки, поскольку она анализирует код на этапе исполнения. (Если анализ оказывается неудачным, ошибка анализа, как обычно, помещается в $@.) В противном случае значение *EXPR* выполняется, как если бы это была маленькая программа Perl. Код выполняется в контексте текущей программы Perl, что означает для него возможность видеть любые охватывающие лексические переменные из окружающей области видимости и сохранение действия установок не локальных переменных по завершении eval – так же, как в случае определений подпрограмм и форматов. Код eval рассматривается как блок, поэтому все переменные с локальной областью видимости, определенные внутри eval, действуют только до завершения eval. (См. my и local.) Как и для всякого кода, в блоке завершающие точка с запятой не требуются.

Вот простая оболочка Perl. Она выводит пользователю приглашение ввести строку произвольного кода Perl, компилирует и выполняет эту строку и выводит ошибки, если они имеют место:

```
print "\Введите какой-нибудь код Perl: ";

while (<STDIN>) {
    eval;
    print $@;
    print "\nВведите еще код Perl: ";
}
```

Вот программа *rename*, осуществляющая групповое переименование файлов с помощью выражения Perl:

```
#!/usr/bin/perl
# rename - изменение имен файлов
$op = shift;
for (@ARGV) {
    $was = $_;
    eval $op;
    die if $@;
    # в следующей строке кода вызывается встроенная функция,
    # а не сценарий с таким же именем
    rename($was,$_) unless $was eq $_;
}
```

Эту программу следует использовать так:

```
$ rename 's/\.orig$//'                *.orig
$ rename 'y/A-Z/a-z/ unless /^Make/'  *
$ rename '$_ .= ".bad"'               *.f
```

Поскольку `eval` перехватывает ошибки, которые иначе оказались бы фатальными, с ее помощью удобно определять, реализованы ли некоторые конкретные возможности (например, `fork` или `symlink`).

Так как `eval` *BLOCK* подвергается синтаксической проверке на этапе компиляции, обо всех синтаксических ошибках сообщается раньше. Поэтому, если вашему коду безразлично и вашим целям одинаково удовлетворят как `eval` *EXPR*, так и `eval` *BLOCK*, предпочтительнее форма *BLOCK*. Например:

```
# сделать деление на ноль не смертельным
eval { $answer = $a / $b; };    warn $@ if $@;

# то же, но менее эффективно при многократном выполнении
eval '$answer = $a / $b';       warn $@ if $@;

# синтаксическая ошибка этапа компиляции (не перехвачена)
eval { $answer = };             # НЕВЕРНО

# синтаксическая ошибка этапа исполнения
eval '$answer =';               # устанавливает $@
```

Здесь код в *BLOCK* должен быть допустимым кодом Perl, чтобы пройти фазу компиляции. Код в *EXPR* не рассматривается до этапа выполнения, поэтому раньше он не вызовет ошибку.

Блок в eval *BLOCK* *не* считается циклом, поэтому команды управления циклом next, last или redo нельзя применять для выхода или перезапуска блока.

exec

$! X T

```
exec PATHNAME LIST
exec LIST
```

Функция exec прекращает выполнение текущей программы и выполняет внешнюю команду, при этом *больше не возвращаясь!!!* Используйте system, а не exec, если хотите восстановить управление после выхода из команды. Вызов функции exec оказывается неуспешным и возвращает «ложь», только если команда не существует *и* если она выполняется непосредственно, а не через командную оболочку системы (обсуждается ниже).

Если имеется только один скалярный аргумент, он проверяется на наличие метасимволов оболочки. Если найдены метасимволы, аргумент целиком передается стандартному интерпретатору команд системы (*/bin/sh* в Unix). Если метасимволов нет, аргумент расщепляется на слова и выполняется непосредственно, поскольку с целью повышения эффективности при этом обходятся накладные расходы обработки оболочкой. Это также дает большие возможности управления восстановлением после ошибки, если программа не существует.

Если в *LIST* больше одного аргумента или *LIST* представляет собой массив, в котором более одного значения, системная оболочка не используется. При этом также обходится какая-либо обработка команды оболочкой. Наличие или отсутствие метасимволов в аргументах не оказывает влияния на этот режим, запускаемый списком, что делает его предпочтительной формой в программах, озабоченных безопасностью, которые не хотели бы разрешить обработку escape-последовательностей оболочкой.

Следующий пример заставляет выполняемую в данный момент программу Perl заменить себя программой *echo*, которая выводит после этого текущий список аргументов:

```
exec 'echo', 'Ваши аргументы: ', @ARGV;
```

Этот пример показывает, что через exec можно запустить не только отдельную программу, но и целый конвейер.

```
exec "sort $outfile | uniq"
    or die "Can't do sort/uniq: $!\n";
```

Обычно возврат из exec не происходит, а если происходит, то возвращает «ложь», и нужно проверить $!, чтобы узнать, что произошло. Учтите, что в прежних версиях Perl функции exec (и system) не очищали выходной буфер, поэтому требовалось включать буферизацию установкой $| для одного или более указателей файлов, чтобы избежать потери вывода в случае exec или смешанного вывода в случае system. Эта ситуация была значительно улучшена в версии Perl 5.6.

Посылая операционной системе запрос на выполнение новой программы в существующем процессе (как это делает функция Perl exec), вы сообщаете системе местонахождение программы, которую нужно выполнить, но также сообщаете новой программе (через первый аргумент) имя, под которым вызвана программа. Обычно сообщаемое имя является просто копией адреса программы, но это не обязательно, т. к. существуют два различных аргумента на уровне языка C. Если это не копия, получается необычный результат, при этом новая программа думает, что она выполняется под некоторым именем, и это имя может отличаться от того пути, где фактически находится программа. Часто это не имеет значения для программы, но иногда программы обращают на это внимание и «надевают» разную личину в зависимости от того, каким считают свое имя. Например, редактор *vi* проверяет, вызвали его как "vi" или как "view". Будучи вызван как "view", он автоматически включает режим «только для чтения», как если бы был вызван с параметром командной строки -R.

Здесь вступает в игру *PATHNAME* — необязательный параметр exec. Синтаксически он занимает позицию косвенного объекта, как указатель файла для print или printf. Он не принимает после себя запятой, поскольку не является в точности частью списка аргументов. (В некотором смысле Perl выбирает противоположный операционной системе подход, предполагая, что важным является первый аргумент, и позволяя изменять имя пути, если оно отличается.) Например:

```
$editor = "/usr/bin/vi";
exec $editor "view", @files      # включение режима только для чтения
    or die "Невозможно выполнить $editor: $!\n";
```

Как и для любого другого косвенного объекта, можно заменить простой скаляр, содержащий имя программы, блоком, содержащим произвольный код, что упрощает предыдущий пример:

```
exec { "/usr/bin/vi" } "view" @files      # включение режима "только для чтения"
    or die "Невозможно выполнить /usr/bin/vi: $!\n";
```

Раньше уже говорилось, что exec рассматривает разрозненный список аргументов как указание на то, что следует избежать обработки оболочкой. Однако в одном случае можно все же споткнуться. Вызов exec (и system тоже) не различает отдельный скалярный аргумент и массив, состоящий только из одного элемента.

```
@args = ("echo surprise");  # всего один элемент в списке
exec @args                  # будут обрабатываться escape-последовательности
     or die "exec: $!";     #   оболочки, потому что @args == 1
```

Чтобы избежать этого, можно использовать синтаксис *PATHNAME*, явно дублируя первый элемент в качестве имени пути, благодаря чему остальные аргументы интерпретируются как список, даже если аргумент всего один:

```
exec { $args[0] } @args   # безопасно даже со списком из одного аргумента
    or die "can't exec @args: $!";
```

Первая версия, без фигурных скобок, запускает программу *echo*, передавая ей в качестве аргумента "surprise". Вторая версия делает не так: она пытается запустить программу, буквально названную *echo surprise*, не находит ее (будем надеяться) и устанавливает переменную $! равной ненулевому значению, указывающему на неудачу.

Поскольку функция exec чаще всего используется вскоре после fork, предполагается, что все, обычно происходящее при окончании процесса Perl, должно быть пропущено. При exec Perl не станет вызывать блоки END или какие-либо методы DESTROY, связанные с любыми объектами. Напротив, ваш дочерний процесс будет в итоге делать ту уборку, которую предположительно должен был делать родительский процесс. (Неплохо бы так в реальной жизни.)

Поскольку очень часто ошибочно используют exec вместо system, Perl выводит предупреждение, если следующая команда отлична от die, warn или exit и используется параметр командной строки **-w** или директива use warnings qw(exec syntax). Если действительно необходимо поместить после exec какую-то другую команду, то можно избежать вывода предупреждения, воспользовавшись любым из следующих стилей:

```
exec ('foo')   or print STDERR "couldn't exec foo: $!";
{ exec ('foo') }; print STDERR "couldn't exec foo: $!";
```

Как показывает вторая строчка, вызов exec как последней команды в блоке освобождает от этого предупреждения.

См. также system.

exists

```
exists EXPR
```

Эта функция возвращает значение «истина», если указанный ключ хеша или индекс массива существует в хеше или массиве. Не имеет значения, является ли соответствующее значение «истиной» или «ложью» и даже определено ли значение.

```
print "True\n"      if         $hash{$key};
print "Defined\n"   if defined $hash{$key};
print "Exists\n"    if exists  $hash{$key};

print "True\n"      if         $array[$index];
print "Defined\n"   if defined $array[$index];
print "Exists\n"    if exists  $array[$index];
```

Элемент может быть «истиной», только если определен, и может быть определен, только если существует, но обратное не обязательно должно выполняться.

Выражение *EXPR* может быть произвольной сложности, при условии, что последней операцией является поиск ключа в хеше или индексирование массива:

```
if (exists $hash{A}{B}{$key}) { ... }
```

Хотя последний элемент не начинает существовать только потому, что было проверено его существование, к промежуточным элементам это относится. Поэтому `$$hash{"A"}` и `$hash{"A"}->{"B"}` оба начнут существовать. Это не функция `exists` *как таковая*; это происходит везде, где используется оператор-стрелка (явно или косвенно):

```
undef $ref;
if (exists $ref->{"Some key"}) { }
print $ref;   # выводит HASH(0x80d3d5c)
```

Даже если элемент `"Some key"` не начал существовать, ранее неопределенная переменная `$ref` начинает вдруг содержать анонимный хеш. Это удивительный пример *самооживления* в том, что на первый (и даже на второй) взгляд не кажется контекстом левой части. Возможно, такой режим будет исправлен в будущих версиях. Чтобы обойти это, можно использовать вложенные вызовы:

```
if ($ref                        and
    exists $ref->[$x]           and
    exists $ref->[$x][$y]       and
    exists $ref->[$x][$y]{$key} and
    exists $ref->[$x][$y]{$key}[2] ) { ... }
```

Если *EXPR* является именем подпрограммы, функция `exists` возвращает значение «истина», если эта подпрограмма объявлена и даже если она еще не определена. Следующий код выводит `"Exists"`:

```
sub flub;
print "Exists\n"     if exists  &flub;
print "Defined\n"    if defined &flub;
```

Вызов `exists` с именем подпрограммы может быть полезен в подпрограмме `AUTOLOAD`, которой нужно знать, требуется ли в некотором пакете, чтобы была определена некоторая подпрограмма. Пакет может указать на это, объявив заглушку `sub` как `flub`.

exit

```
exit EXPR
exit
```

Эта функция вычисляет *EXPR* как целое число и немедленно выходит с этим значением в качестве статуса ошибки программы. Если выражение *EXPR* опущено, функция выходит со статусом `0` (означающим отсутствие ошибки). Вот фрагмент кода, позволяющий пользователю выйти из программы, введя `x` или `X`:

```
$ans = <STDIN>;
exit if $ans =~ /^[Xx]/;
```

Не следует применять exit для прерывания выполнения подпрограммы, если существует вероятность, что кто-нибудь захочет перехватывать происшедшие ошибки. Вызывайте вместо нее die, которую можно перехватывать с помощью eval. Либо используйте оболочки die из модуля Carp, например croak или confess.

Мы сказали, что функция exit осуществляет немедленный выход, но это откровенная неправда. Она выходит, как только это станет возможным, но сначала вызывает все имеющиеся программы END для обработки при выходе. Эти программы не могут прервать выход, но могут изменить итоговое значение завершения путем установки переменной $?. Аналогично любой класс, в котором определен метод DESTROY, вызовет его для всех своих объектов, прежде чем произойдет действительный выход из программы. Если действительно требуется обойти обработку при выходе, можно вызвать функцию _exit из модуля POSIX и избежать обработки всех END и деструкторов. А если POSIX отсутствует, можно вызвать exec "/bin/false" или сделать что-либо аналогичное.

exp

```
exp EXPR
exp
```

Эта функция возвращает *e* в степени *EXPR*. Чтобы получить значение *e*, вызывайте exp(1). Для общего возведения в степень по любому основанию используйте оператор **, украденный нами из FORTRAN:

```
use Math::Complex;
print -exp(1) ** (i * pi);   # выводит 1
```

fcntl

```
fcntl FILEHANDLE, FUNCTION, SCALAR
```

Эта функция вызывает функции управления файлами вашей операционной системы, документированные на странице руководства *fcntl*(2). Перед вызовом fcntl, вероятно, нужно сначала сказать:

```
use Fcntl;
```

чтобы загрузить правильные определения констант.

SCALAR можно читать или записывать (или и то и другое) в зависимости от *FUNCTION*. Указатель на строковое значение *SCALAR* будет передан в качестве третьего аргумента фактического вызова *fcntl*. (Если *SCALAR* не имеет строкового значения, но имеет числовое значение, то будет передано непосредственно это значение, а не указатель на строковое значение.) См. в описании модуля Fcntl наиболее употребительные допустимые значения для *FUNCTION*.

Функция fcntl создаст исключительную ситуацию при вызове в системе, в которой не реализована *fcntl*(2). В системах, в которых она реализована,

можно, например, модифицировать флаги закрытия при exec (если вы не хотите использовать переменную $^F ($SYSTEM_FD_MAX)), модифицировать не блокирующие флаги ввода-вывода, эмулировать функцию *lockf*(3) и организовывать прием сигнала SIGIO при наступлении ввода-вывода.

Вот пример установки указателя файла с именем REMOTE, не блокирующим на системном уровне. В результате любая операция ввода немедленно возвращается, если нет ничего для чтения из канала, сокета или последовательной линии, которая иначе была бы блокирующей. При этом также операции вывода вместо обычного блокирования возвращают статус неуспеха. (В этом случае вам, вероятно, придется также уладить дело с $|.)

```
use Fcntl qw(F_GETFL F_SETFL O_NONBLOCK);

$flags = fcntl(REMOTE, F_GETFL, 0)
            or die "Невозможно получить флаги для сокета: $!\n";

$flags = fcntl(REMOTE, F_SETFL, $flags | O_NONBLOCK)
            or die "Невозможно установить флаги для сокета: $!\n";
```

Значения, возвращаемые fcntl (и ioctl):

Значения, возвращаемые системным вызовом	Значения, возвращаемые Perl
-1	undef
0	строка "0 but true"
все прочее	это число

Таким образом, Perl возвращает значение «истина» при успехе и «ложь» при неудаче, хотя сохраняется возможность определить фактическое значение, возвращенное операционной системой:

```
$retval = fcntl(...) || -1;
printf "fcntl фактически возвратила %d\n", $retval;
```

Здесь даже строка "0 but true" выводит 0 благодаря формату %d. Эта строка истинна в булевом контексте и принимает значение 0 в числовом контексте. (Она также «с удобствами» избавляет от обычных предупреждений о недопустимом числовом преобразовании.)

fileno

```
fileno FILEHANDLE
```

Эта функция возвращает дескриптор файла (file descriptor), лежащий в основе указателя файла (filehandle). Если указатель не открыт, fileno возвращает undef. *Дескриптор файла* – это маленькое неотрицательное целое число, например 0 или 1, в отличие от таких указателей файлов, как STDIN и STDOUT, которые представляют собой символы. К несчастью, операционной системе ничего не известно о ваших замечательных символах. Она представ-

ляет себе открытые файлы только как эти маленькие номера файлов, и хотя Perl обычно автоматически выполняет трансляцию, иногда требуется знать фактический дескриптор файла.

Поэтому, например, функция `fileno` полезна для создания битовых массивов для `select` и передачи неким неясным системным вызовам, если реализована *syscall*(2). С ее помощью также можно дополнительно проверить, возвратила ли функция `open` нужный дескриптор файла, и определить, не используется ли один и тот же системный дескриптор файла двумя указателями файла.

```
if (fileno(THIS) == fileno(THAT)) {
    print "THIS и THAT являются дубликатами \n";
}
```

Если *FILEHANDLE* является выражением, его значение принимается в качестве косвенного указателя файла, обычно его имени или чего-то похожего на объект указателя файла.

Одно предостережение: не полагайтесь на сохранение связи между указателем файла Perl и числовым дескриптором файла на протяжении всего времени работы программы. Если файл закрывается и открывается заново, дескриптор может измениться. Perl берет на себя некоторый труд, стараясь сделать так, чтобы некоторые дескрипторы не потерялись, если вызов `open` для них окажется неудачным, но делает это только для дескрипторов файлов, которые не превосходят текущего значения специальной переменной `$^F` (`$SYSTEM_FD_MAX`), по умолчанию равного 2. Хотя указатели файлов `STDIN`, `STDOUT` и `STDERR` вначале имеют дескрипторы файлов 0, 1 и 2 (стандартное соглашение Unix), но даже они могут измениться, если начать закрывать и открывать их достаточно энергично. Неприятностей с 0, 1 и 2 быть не может, если открывать их заново немедленно после закрытия. Основное правило в Unix – выбирать наименьший имеющийся дескриптор, а это именно тот, который был только что закрыт.

flock

```
flock FILEHANDLE, OPERATION
```

Функция `flock` является переносимым интерфейсом Perl для блокировки файлов, хотя он блокирует только целые файлы, а не записи. Эта функция управляет блокировками файла, связанного с *FILEHANDLE*, возвращая «истину» в случае успеха и «ложь» в противном случае. Чтобы исключить возможность потери данных, Perl очищает буфер *FILEHANDLE* перед блокированием или разблокированием его. Perl может реализовать `flock` посредством *flock*(2), *fcntl*(2), *lockf*(3) или другого специфического для платформы механизма блокировки, но если ни один из них не доступен, то вызов `flock` приводит к возникновению исключительной ситуации. См. раздел «Блокировка файлов» главы 16.

Аргумент *OPERATION* может принимать значения `LOCK_SH`, `LOCK_EX` или `LOCK_UN`, которые могут быть соединены через ИЛИ с `LOCK_NB`. Эти константы традици-

онно имеют значения 1, 2, 8 и 4, но можно обращаться к ним по символическим именам, если импортировать их из модуля Fcntl отдельно или как группу, используя тег :flock.

LOCK_SH запрашивает блокировку с совместным доступом, поэтому обычно она применяется для чтения. LOCK_EX запрашивает монопольную блокировку, поэтому обычно предназначается для записи. LOCK_UN освобождает ранее запрошенную блокировку; закрытие файла тоже снимает все блокировки. Если бит LOCK_NB используется с LOCK_SH или LOCK_EX, то flock возвращается немедленно, не ожидая недоступной блокировки. Проверяйте статус возврата, чтобы определить, получена ли запрашиваемая блокировка. Если LOCK_NB не используется, можно бесконечно ожидать, пока будет предоставлена блокировка.

Другой неочевидной, но традиционной стороной flock является то, что осуществляемые ею блокировки – *чисто рекомендательные*. Дискреционные блокировки характеризуются большей гибкостью, но дают меньше гарантий, чем обязательные. Это значит, что файлы, заблокированные flock, могут быть модифицированы программами, которые сами не вызывают flock. Машины, останавливающиеся на красный свет, хорошо уживаются друг с другом, но не с машинами, которые не останавливаются на красный свет. Будьте бдительны при езде.

Некоторые реализации flock не могут блокировать файлы в сетевой среде. Хотя теоретически можно применять с этой целью более специфический для системы fcntl, но присяжные (удалившись на совещание по этому делу лет десять назад) все еще не решили, надежно ли это (и даже может ли это быть надежным).

Вот программа для добавления корреспонденции в почтовый ящик в системах Unix, использующая *flock*(2) для блокировки почтовых ящиков:

```
use Fcntl qw/:flock/;       # импорт констант LOCK_*
sub mylock {
    flock(MBOX, LOCK_EX)
        or die "невозможно блокировать mailbox: $!";
    # на случай, если кто-то что-то дописл, пока мы ждали,
    # и наш буфер stdio рассинхронизировался
    seek(MBOX, 0, 2)
        or die "невозможно seek в конец mailbox: $!";
}
open(MBOX, ">>/usr/spool/mail/$ENV{'USER'}")
    or die "невозможно открыть mailbox: $!";

mylock();
print MBOX $msg, "\n\n";
close MBOX
    or die "невозможно закрыть mailbox: $!";
```

В системах, реально поддерживающих системный вызов *flock*(2), блокировки наследуются в вызовах fork. Другим реализациям не так повезло, и при

ветвлении они, скорее всего, теряют блокировки. См. также в разделе «Блокировка файлов» главы 16 другие примеры использования `flock`.

fork

```
fork
```

Эта функция делает из одного процесса два с помощью системного вызова *fork*(2). В случае успеха функция возвращает ID нового дочернего процесса в родительский процесс и 0 в дочерний процесс. Если у системы недостаточно ресурсов для размещения нового процесса, вызов оказывается неудачным и возвращает `undef`. Дескрипторы файлов (и иногда блокировки этих дескрипторов) используются совместно, а все остальное копируется или, по крайней мере, выглядит таким образом.

В версиях Perl до 5.6 неочищенные буферы остаются неочищенными в обоих процессах, что означает потребность в установке `$|` для одного или более указателей файлов ранее в программе, чтобы избежать дублирования вывода.

Почти безопасный способ запустить дочерний процесс при проверке ошибок невозможности ветвления может быть таким:

```
use Errno qw(EAGAIN);
FORK: {
    if ($pid = fork) {
        # здесь родительский
        # pid дочернего процесса в $pid
    }
    elsif (defined $pid) { # $pid равен нулю, если определен
        # здесь дочерний
        # pid подительского процесса доступен через getppid
    }
    elsif ($! == EAGAIN) {
        # EAGAIN является предположительно восстанавливаемой ошибкой ветвления
        sleep 5;
        redo FORK;
    }
    else {
        # фатальная ошибка ветвления
        die "Разделение на процессы невозможно: $!\n";
    }
}
```

Такие предосторожности не обязательны для операций, которые неявно выполняют *fork*(2), например, `system`, обратный штрих или открытие процесса как указателя файла, потому что Perl автоматически повторяет порождение дочернего процесса при временной неудаче, когда выполняет `fork` вместо вас. Следите за тем, чтобы заканчивать код дочернего процесса с помощью `exit`, иначе дочерний процесс непреднамеренно выйдет из условно выполняемого блока и начнет выполнять код, предназначенный только для родительского процесса.

Выполняя fork и не дожидаясь завершения своих дочерних процессов, вы накопите «зомби» (завершившиеся процессы, которых не ожидали родители). В некоторых системах этого можно избежать путем установки $SIG{CHLD} в "IGNORE"; в большинстве случаев нужно выполнять wait для умирающих дочерних процессов. Примеры см. в описании функции wait, а подробности о SIGCHLD – в разделе «Сигналы» главы 16.

Если ответвившийся процесс наследует дескрипторы системных файлов, такие как STDIN и STDOUT, соединенные с удаленным каналом или сокетом, может потребоваться переоткрыть их в дочернем процессе на */dev/null*. Дело в том, что даже при завершении родительского процесса дочерний процесс продолжает пользоваться своими копиями этих указателей файлов. Удаленный сервер (например, сценарий CGI или фоновое задание, запущенное в удаленной оболочке) окажется зависшим, т. к. будет ожидать закрытия всех копий. Повторное открытие системных указателей куда-либо еще исправит положение.

В большинстве систем, поддерживающих *fork*(2), большие старания были приложены, чтобы сделать ее очень эффективной (например, использование технологии копирования при записи (copy-on-write) для страниц данных), обусловившие доминирование этой парадигмы многозадачности в последние десятилетия. Трудно рассчитывать, что функция fork реализована эффективно (если реализована вообще) на системах, не имеющих сходства с Unix. Например, Perl 5.6 эмулирует правильную fork даже на системах Microsoft, но каких-либо утверждений относительно производительности в этом случае сделать нельзя. Возможно, бо́льшая удача ждет вас в случае применения модуля Win32::Process.

format

```
format NAME =
    picture line
    value list
    ...
.
```

Эта функция объявляет именованную последовательность строк шаблона (picture lines) и связанных величин для использования в функции write. Если *NAME* опущено, то именем по умолчанию становится STDOUT, которое оказывается именем формата по умолчанию для указателя файла STDOUT. Поскольку, подобно объявлению sub, это – глобальное для пакета объявление, совершаемое на этапе компиляции, то любые переменные, присутствующие в списке значений, должны быть видимы в точке объявления формата. Это значит, что переменные с лексической областью видимости должны быть объявлены выше в файле, а переменные с динамической областью видимости просто должны быть установлены во время вызова write. Вот пример (в котором предполагается, что мы уже рассчитали $cost и $quantity):

```
my $str = "widget";                # Переменная с лексической областью видимости.

format Nice_Output =
Test: @<<<<<<<< @||||| @>>>>>
      $str,     $%,    '$' . int($num)
.

local $~ = "Nice_Output";          # Выбираем наш формат.
local $num = $cost * $quantity;    # Переменная с динамической областью видимости.
write;
```

Как и указатели файлов, имена форматов являются идентификаторами, существующими в таблице символов (пакете), и могут полностью квалифицироваться именем пакета. Вместе с переменными типа typeglob в таблице символов форматы располагаются в собственном пространстве имен, отличном от указателей файлов, указателей каталогов, скаляров, массивов, хешей и подпрограмм. Однако, как и на эти шесть типов, на формат с именем `Whatever` окажет воздействие вызов `local`, примененный к `*Whatever`. Иными словами, формат – это еще одна штуковина, содержащаяся в переменной typeglob, независимая от других штуковин.

Раздел «Переменные форматов» главы 7 «Форматы» содержит многочисленные детали и примеры их использования. Глава 28 описывает внутренние специфические для форматов переменные, а модули `English` и `IO::Handle` предоставляют более легкий доступ к ним.

formline

```
formline PICTURE, LIST
```

Это внутренняя функция, которую вызывает `format`, хотя можно вызвать ее и самостоятельно. Она всегда возвращает «истину». Она форматирует список значений согласно содержимому шаблона *PICTURE*, помещая вывод в накопитель форматированного вывода, `$^A` (или `$ACCUMULATOR`, если используется модуль `English`). В конечном счете, при выполнении `write` содержимое `$^A` выводится в какой-нибудь указатель файла, но можно прочесть `$^A` самостоятельно, а затем установить `$^A` в `""`. Формат обычно осуществляет один вызов `formline` для каждой строки формы, но сама функция `formline` не смотрит, сколько символов перевода строки имеется в *PICTURE*. Это значит, что маркеры `~` и `~~` будут рассматривать *PICTURE* в целом как одну строку. Поэтому может потребоваться несколько вызовов `formline` для реализации формата одной записи, как это и делает компилятор формата.

Будьте осторожны, если заключаете шаблон в двойные кавычки, поскольку символ `@` может быть принят за начало имени массива. Примеры использования см. в разделе «Форматы» главы 6 «Подпрограммы».

getc

```
getc FILEHANDLE
getc
```

Эта функция возвращает очередной байт из входного файла, прикрепленного к *FILEHANDLE*. Она возвращает undef в позиции конца файла или при возникновении ошибки ввода-вывода. Если *FILEHANDLE* опущен, чтение производится из STDIN.

Эта функция несколько медлительна, но иногда полезна для ввода одиночных символов (на самом деле, байтов) с клавиатуры – при условии, что ввод с клавиатуры не буферизован. Эта функция запрашивает небуферизованный ввод из стандартной библиотеки ввода-вывода. К несчастью, стандартная библиотека ввода-вывода не настолько стандартна, чтобы обеспечить переносимый способ сказать операционной системе, чтобы она предоставила небуферизованный ввод с клавиатуры для стандартной системы ввода-вывода. Для этого надо оказаться немного умнее (ее) и при этом учитывать зависимость от операционной системы. В Unix можно поступить так:

```
if ($BSD_STYLE) {
  system "stty cbreak </dev/tty >/dev/tty 2>&1";
} else {
  system "stty", "-icanon", "eol", "\001";
}

$key = getc;

if ($BSD_STYLE) {
  system "stty -cbreak </dev/tty >/dev/tty 2>&1";
} else {
  system "stty", "icanon", "eol", "^@"; # ASCII NUL
}
print "\n";
```

Этот код помещает очередной символ (байт), введенный с терминала, в строку $key. Если программа *stty* поддерживает такую опцию, как cbreak, то потребуется использовать этот код, если $BSD_STYLE имеет значение «истина». В противном случае придется его использовать, когда она принимает значение «ложь». Определение опций *stty*(1) оставлено в качестве упражнения для читателя.

Модуль POSIX предоставляет лучше переносимую версию с функцией POSIX::getattr. Более переносимый и гибкий подход можно найти в модуле Term::ReadKey на ближайшем к вам сайте CPAN.

getgrent

```
getgrent
setgrent
endgrent
```

Эти программы обходят ваш файл */etc/group* (или чей-то чужой */etc/group*, если он находится где-то на сервере). Возвращаемым значением getgrent в списковом контексте служит

```
($name, $passwd, $gid, $members)
```

где $members содержит список разделенных пробелами имен регистрации членов группы. Организовать хеш для перевода имен групп в GID можно так:

```
while (($name, $passwd, $gid) = getgrent) {
    $gid{$name} = $gid;
}
```

В скалярном контексте getgrent возвращает только имя группы. Стандартный модуль User::grent поддерживает интерфейс к этой функции через имена. См. *getgrent*(3).

getgrgid

X U

```
getgrgid GID
```

Эта функция ищет запись в файле групп по номеру группы. В списковом контексте возвращаемым значением является

```
($name, $passwd, $gid, $members)
```

где $members содержит список разделенных пробелами имен регистрации членов группы. Если это необходимо делать неоднократно, подумайте о кэшировании данных в хеше с помощью getgrent.

В скалярном контексте getgrgid возвращает только имя группы. Модуль User::grent поддерживает интерфейс к этой функции через имена. См. *getgrgid*(3).

getgrnam

X U

```
getgrnam NAME
```

Эта функция ищет запись в файле групп по имени группы. В списковом контексте возвращаемым значением является

```
($name, $passwd, $gid, $members)
```

где $members содержит список разделенных пробелами имен регистрации членов группы. Если приходится делать это неоднократно, подумайте о кэшировании данных в хеше с помощью getgrent.

В скалярном контексте getgrnam возвращает только числовой ID группы. Модуль User::grent поддерживает интерфейс к этой функции через имена. См. *getgrnam*(3).

gethostbyaddr

X U

```
gethostbyaddr ADDR, ADDRTYPE
```

Эта функция транслирует адреса в имена (и в альтернативные адреса). *ADDR* должен быть упакованным двоичным сетевым адресом, а в роли *ADDRTYPE*

на практике обычно выступает AF_INET (из модуля Socket). Возвращаемым в списковом контексте значением является

```
($name, $aliases, $addrtype, $length, @addrs) =
    gethostbyaddr($packed_binary_address, $addrtype);
```

где @addrs является списком упакованных двоичных адресов. В домене Интернета каждый адрес (исторически) имеет длину четыре байта и может быть распакован, например, так:

```
($a, $b, $c, $d) = unpack('C4', $addrs[0]);
```

Альтернативным вариантом является непосредственное преобразование в нотацию вектора с точками с помощью модификатора v для sprintf:

```
$dots = sprintf "%vd", $addrs[0];
```

Функцию inet_ntoa из модуля Socket удобно использовать при создании версии для вывода через print. Этот подход станет важным, если мы все когда-либо перейдем на IPv6.

```
use Socket;
$printable_address = inet_ntoa($addrs[0]);
```

В скалярном контексте gethostbyaddr возвращает только имя хоста.

Чтобы создать *ADDR* из вектора с точками, сделайте так:

```
use Socket;
$ipaddr = inet_aton("127.0.0.1");       # localhost
$claimed_hostname = gethostbyaddr($ipaddr, AF_INET);
```

Интересно, что в версии Perl 5.6 можно пропустить inet_aton()и придерживаться новой нотации v-строк, придуманной для номеров версий, но которая может работать и с IP-адресами:

```
$ipaddr = v127.0.0.1;
```

Дополнительные примеры см. в разделе «Сокеты» главы 16. Модуль Net::hostent обеспечивает поддержку интерфейса к этой функции через имена. См. *gethostbyaddr*(3).

gethostbyname

```
gethostbyname NAME
```

Эта функция транслирует сетевое имя хоста в соответствующие адреса (и другие имена). В списковом контексте возвращается значение:

```
($name, $aliases, $addrtype, $length, @addrs) =
    gethostbyname($remote_hostname);
```

где @addrs – список «сырых» адресов. В домене Интернета каждый адрес (исторически) имеет длину четыре байта и может быть распакован, например, так:

```
($a, $b, $c, $d) = unpack('C4', $addrs[0]);
```

Можно непосредственно преобразовать в нотацию вектора с точками с помощью модификатора v для sprintf:

```
$dots = sprintf "%vd", $addrs[0];
```

В скалярном контексте gethostbyname возвращает только адрес хоста:

```
use Socket;
$ipaddr = gethostbyname($remote_host);
printf "%s has address %s\n",
    $remote_host, inet_ntoa($ipaddr);
```

О другом подходе рассказано в разделе «Сокеты» главы 16. Модуль Net::hostent обеспечивает поддержку интерфейса к этой функции через имена. См. также *gethostbyname*(3).

gethostent

```
gethostent
sethostent STAYOPEN
endhostent
```

Эти функции осуществляют итерацию по файлу */etc/hosts* и возвращают каждый раз по одной записи. Функция gethostent возвращает значение:

```
($name, $aliases, $addrtype, $length, @addrs)
```

где @addrs является списком сырых адресов. В домене Интернета каждый адрес (исторически) имеет длину четыре байта и может быть распакован, например, так:

```
($a, $b, $c, $d) = unpack('C4', $addrs[0]);
```

Сценарии, содержащие вызов gethostent, не следует считать переносимыми. Если машина использует сервер имен, ей придется опросить большую часть Интернета, чтобы удовлетворить запрос всех адресов всех машин на свете. Поэтому gethostent не реализована на таких машинах. Другие подробности можно найти в описании *gethostent*(3).

Модуль Net::hostent обеспечивает поддержку интерфейса к этой функции через имена.

getlogin

```
getlogin
```

Эта функция возвращает текущее имя регистрации, если оно найдено. В Unix-системах оно считывается из файла *utmp*(5). Если она возвращает «ложь», используйте вместо нее getpwuid. Например:

```
$login = getlogin() || (getpwuid($<))[0] || "Нарушитель!!";
```

getnetbyaddr

```
getnetbyaddr ADDR, ADDRTYPE
```

Эта функция транслирует сетевой адрес в соответствующее сетевое имя или имена. В списковом контексте функция возвращает значение:

```
use Socket;
($name, $aliases, $addrtype, $net) = getnetbyaddr(127, AF_INET);
```

В скалярном контексте getnetbyaddr возвращает сетевое имя. Модуль Net::netent обеспечивает поддержку интерфейса к этой функции через имена. См. *getnetbyaddr* (3).

getnetbyname

```
getnetbyname NAME
```

Эта функция транслирует сетевое имя в соответствующий сетевой адрес. В списковом контексте возвращает значение:

```
($name, $aliases, $addrtype, $net) = getnetbyname("loopback");
```

В скалярном контексте getnetbyname возвращает только сетевой адрес. Модуль Net::netent обеспечивает поддержку интерфейса к этой функции через имена. См. *getnetbyname*(3).

getnetent

```
getnetent
setnetent STAYOPEN
endnetent
```

Эти функции осуществляют итерацию по файлу */etc/networks*. В списковом контексте возвращают значение:

```
($name, $aliases, $addrtype, $net) = getnetent();
```

В скалярном контексте getnetent возвращает только сетевое имя. Модуль Net::netent обеспечивает поддержку интерфейса к этой функции через имена. См. *getnetent*(3).

Концепция сетевых имен кажется в наше время довольно странной; большинство IP-адресов находится в неименованных (и неименуемых) подсетях.

getpeername

```
getpeername SOCKET
```

Эта функция возвращает упакованный адрес сокета противоположного конца соединения SOCKET. Например:

```
use Socket;
$hersockaddr      = getpeername SOCK;
($port, $heraddr) = sockaddr_in($hersockaddr);
$herhostname      = gethostbyaddr($heraddr, AF_INET);
$herstraddr       = inet_ntoa($heraddr);
```

getpgrp

$! X U

```
getpgrp PID
```

Эта функция возвращает текущую группу процессов для указанного *PID* (в ожидании текущего процесса используйте *PID*, равный 0). Вызов getpgrp создаст исключительную ситуацию на машине, где не реализован *getpgrp*(2). Если *PID* опущен, функция возвращает группу процессов текущего процесса (то же, что при *PID*, равном 0). В системах, реализующих этот оператор через системный вызов POSIX *getpgrp*(2), *PID* должен быть опущен или задан равным 0.

getppid

X U

Эта функция возвращает ID родительского процесса. В типичной системе Unix, когда ID родительского процесса изменяется на 1, это означает, что ваш родительский процесс умер и вас приняла программа *init*(8).

getpriority

$! X U

```
getpriority WHICH, WHO
```

Эта функция возвращает текущий приоритет процесса, группы процессов или пользователя. См. *getpriority*(2). Вызов getpriority создаст исключительную ситуацию на машине, где не реализован *getpriority*(2).

Модуль BSD::Resource в CPAN предоставляет более удобный интерфейс, в том числе символические константы PRIO_PROCESS, PRIO_PGRP и PRIO_USER, передаваемые в аргументе *WHICH*. Хотя обычно они равны 0, 1 и 2 соответственно, никогда нельзя знать, что может произойти в темных глубинах файлов #include C.

Значение 0 для *WHO* означает текущий процесс, группу процессов или пользователя, поэтому получить приоритет текущего процесса можно так:

```
$curprio = getpriority(0, 0);
```

getprotobyname

X U

```
getprotobyname NAME
```

Эта функция транслирует имя протокола в соответствующий ему номер. В списковом контексте возвращается следующее значение:

```
($name, $aliases, $protocol_number) = getprotobyname("tcp");
```

При вызове в скалярном контексте `getprotobyname` возвращает только номер протокола. Модуль `Net::proto` поддерживает интерфейс к этой функции через имена. См. *getprotobyname*(3).

getprotobynumber

X U

```
getprotobynumber NUMBER
```

Эта функция транслирует номер протокола в соответствующее имя. В списковом контексте возвращается следующее значение:

```
($name, $aliases, $protocol_number) = getprotobynumber(6);
```

При вызове в скалярном контексте `getprotobynumber` возвращает только имя протокола. Модуль `Net::proto` поддерживает интерфейс к этой функции через имена. См. *getprotobynumber* (3).

getprotoent

X U

```
getprotoent
setprotoent STAYOPEN
endprotoent
```

Эти функции осуществляют итерацию файла */etc/protocols*. В списковом контексте `getprotoent` возвращает значение:

```
($name, $aliases, $protocol_number) = getprotoent();
```

При вызове в скалярном контексте `getprotoent` возвращает только имя протокола. Модуль `Net::proto` поддерживает интерфейс к этой функции через имена. См. *getprotent*(3).

getpwent

T X U

```
getpwent
setpwent
endpwent
```

Эти функции осуществляют итерацию файла */etc/passwd*, хотя в ней может участвовать файл */etc/shadow*, если вы – суперпользователь и применяете теневые пароли, либо NIS (в девичестве YP) или NIS+, если вы работаете с одним из них. В списковом контексте возвращается следующее значение:

```
($name,$passwd,$uid,$gid,$quota,$comment,$gcos,$dir,$shell) = getpwent();
```

На некоторых машинах поля квоты и комментария могут использоваться с другими целями, чем указывающие на то имена, но остальные поля всегда одинаковы. Чтобы создать хеш для трансляции имен регистрации в UID, сделайте так:

```
while (($name, $passwd, $uid) = getpwent()) {
    $uid{$name} = $uid;
}
```

В скалярном контексте getpwent возвращает только имя пользователя. Модуль User::pwent поддерживает интерфейс к этой функции через имена. См. *getpwent*(3).

getpwnam

T X U

```
getpwnam NAME
```

Эта функция транслирует имя пользователя в соответствующую запись файла */etc/passwd*. В списковом контексте возвращается следующее значение:

```
($name,$passwd,$uid,$gid,$quota,$comment,$gcos,$dir,$shell) = getpwnam("daemon");
```

В системах, поддерживающих теневые пароли, нужно быть суперпользователем, чтобы извлечь настоящий пароль. Ваша библиотека C должна обратить внимание на то, что у вас есть достаточные права, и открыть файл */etc/shadow* (или где там еще хранится файл *shadow*). По крайней мере, так она должна работать. Если библиотека C слишком глупа, чтобы это заметить, это попытается сделать Perl.

Для многократного поиска можно кэшировать данные с помощью getpwent.

В скалярном контексте getpwnam возвращает только числовой ID пользователя. Модуль User::pwent поддерживает интерфейс к этой функции через имена. См. *getpwnam*(3) и *passwd*(5).

getpwuid

T X U

```
getpwuid UID
```

Эта функция транслирует числовой ID пользователя в соответствующую запись файла */etc/passwd*. В списковом контексте возвращается следующее значение:

```
($name,$passwd,$uid,$gid,$quota,$comment,$gcos,$dir,$shell) = getpwuid(2);
```

Для многократного поиска можно кэшировать данные с помощью getpwent.

В скалярном контексте getpwuid возвращает имя пользователя. Модуль User::pwent поддерживает интерфейс к этой функции через имена. См. *getpwnam*(3) и *passwd*(5).

getservbyname

X U

```
getservbyname NAME, PROTO
```

Эта функция транслирует имя сервиса (порта) в соответствующий номер порта. *PROTO* представляет собой имя протокола, например "tcp". В списковом контексте возвращается следующее значение:

```
($name, $aliases, $port_number, $protocol_name) = getservbyname("www", "tcp");
```

В скалярном контексте getservbyname возвращает только номер порта сервиса. Модуль Net::servent поддерживает интерфейс к этой функции через имена. См. *getservbyname*(3).

getservbyport

X U

```
getservbyport PORT, PROTO
```

Эта функция транслирует номер сервиса (порта) в соответствующее имя. *PROTO* является именем протокола, например "tcp". В списковом контексте возвращается следующее значение:

```
($name, $aliases, $port_number, $protocol_name) = getservbyport(80, "tcp");
```

В скалярном контексте getservbyport возвращает только имя сервиса. Модуль Net::servent поддерживает интерфейс к этой функции через имена. См. *getservbyport*(3).

getservent

X U

```
getservent
setservent STAYOPEN
endservent
```

Эта функция осуществляет итерацию по файлу */etc/services* или эквивалентному ему. В списковом контексте возвращается следующее значение:

```
($name, $aliases, $port_number, $protocol_name) = getservent();
```

В скалярном контексте getservent возвращает только имя сервиса порта. Модуль Net::servent поддерживает интерфейс к этой функции через имена. См. *getservent*(3).

getsockname

$! X ARG X U

```
getsockname SOCKET
```

Эта функция возвращает упакованный адрес сокета данного конца соединения *SOCKET*. (А почему вы до сих пор не знаете свой собственный адрес? Может быть, потому, что вы привязали адрес, содержащий групповые символы, к сокету сервера перед выполнением accept и теперь хотите узнать, какой интерфейс кто-то использовал для соединения с вами. Либо сокет был передан вам родительским процессом, например *inetd*.)

```
use Socket;
$mysockaddr = getsockname(SOCK);
($port, $myaddr) = sockaddr_in($mysockaddr);
$myname = gethostbyaddr($myaddr,AF_INET);
printf "I am %s [%vd]\n", $myname, $myaddr;
```

getsockopt

$! X ARG X U

```
getsockopt SOCKET, LEVEL, OPTNAME
```

Эта функция возвращает запрошенную опцию сокета или `undef` в случае ошибки. См. дополнительные сведения в `setsockopt`.

glob

$_ $@ T X T

```
glob EXPR
glob
```

Эта функция возвращает значения *EXPR* с расширениями (expansions) имен файлов, как это делает оболочка (shell). Это внутренняя функция, реализующая оператор `<*>`.

В силу исторических причин алгоритм соответствует стилю расширения в *csh*(1), а не в оболочке Борна. Версии Perl до 5.6 использовали внешний процесс, но 5.6 и выше осуществляют внутренний поиск файлов по шаблону. Файлы, первым символом имени которых является точка «`.`», игнорируются, если только этот символ не задан явно. Звездочка «`*`» соответствует любой последовательности любых символов (в том числе отсутствию символов). Вопросительный знак «`?`» соответствует любому одному символу. Последовательность в квадратных скобках «`[...]`» задает простой класс символов, например «`[chyO-9]`». Классы символов могут отрицаться с помощью циркумфлекса, как в «`*.[^oa]`», что ищет любые файлы, начинающиеся не с точки, в именах которых есть точка, за которой следует один символ, отличный от «a» и «o» и являющийся последним в имени. Тильда «`~`» расширяется в исходный каталог пользователя, как в «`~/.*rc`» для всех файлов «rc» текущего пользователя, или в «`~jane/Mail/*`» для всех почтовых файлов Jane. Можно использовать фигурные скобки для чередования, как в «`~/.{mail,ex,csh,twm,}rc`» для получения этих конкретных rc-файлов.

Для того чтобы осуществлять поиск имен файлов, которые могут содержать пробельные символы, потребуется работать с модулем `File::Glob` непосредственно, поскольку `glob` исстари использует пробелы для разделения нескольких шаблонов, например `<*.c *.h>`. За деталями обратитесь к описанию модуля `File::Glob` в главе 32. При вызове `glob` (или оператора `<*>`) этот модуль автоматически загружается через `use`, поэтому если модуль каким-либо таинственным образом испарится из вашей библиотеки, будет возбуждена исключительная ситуация.

При вызове `open` Perl не расширяет групповые символы, в том числе тильду. Нужно сначала выполнить `glob`.

```
open(MAILRC, "~/.mailrc")                # НЕВЕРНО: тильда относится к оболочке
    or die "невозможно открыть ~/.mailrc: $!";

open(MAILRC, (glob("~/.mailrc"))[0])     # сначала расширить тильду
    or die "невозможно открыть ~/.mailrc: $!";
```

Функция glob не связана с понятием переменных typeglob в Perl, если не считать того, что в обоих случаях * служит для представления группы объектов.

См. также раздел «Оператор поиска файлов по шаблону имени» главы 2.

gmtime

```
gmtime EXPR
gmtime
```

Эта функция преобразует время из формата, возвращаемого функцией time, в список из девяти элементов, представляющих время для часового пояса Гринвича (также известное как GMT или UTC, или даже Zulu в некоторых странах, исключая, как ни странно, саму культуру Zulu). Обычно она (функция, а не культура) используется так:

```
#  0    1    2     3     4    5     6     7     8
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = gmtime;
```

Если, как в данном случае, *EXPR* опущено, осуществляется вызов gmtime(time()). Модуль из библиотеки Perl Time::Local содержит подпрограмму timegm, которая может преобразовать список обратно в значение времени.

Все элементы списка являются числами и поступают прямо из struct tm (это программная структура C – не волнуйтесь). В частности это означает, что $mon имеет диапазон 0..11, при этом январю соответствует номер 0, $wday имеет диапазон 0..6, а воскресенью соответствует номер 0. Можно запомнить, какие величины начинают нумероваться с нуля, поскольку именно они всегда выступают в качестве индексов в нумерующихся с нуля массивах, содержащих названия месяцев и дней недели.

Например, получить текущий месяц в Лондоне можно так:

```
$london_month = (qw(Jan Feb Mar Apr May Jun
                    Jul Aug Sep Oct Nov Dec))[(gmtime)[4]];
```

$year – это число лет, прошедших после 1900; таким образом, в 2023 году $year будет иметь значение 123, а *не* просто 23. Чтобы получить 4-значный год, просто скажите $year + 1900. Чтобы получить 2-значный год (например, «01» в 2001 году), используйте sprintf("%02d", $year % 100).

В скалярном контексте gmtime возвращает строку в стиле *ctime*(3), основанную на значении времени GMT. Модуль Time::gmtime поддерживает интерфейс к этой функции через имена. См. также в POSIX::strftime() о подходе к форматированию времени с большим разрешением.

Это скалярное значение *не* зависит от национальных настроек, а является встроенным для Perl. См. также модуль Time::Local и функции *strftime*(3) и *mktime*(3), имеющиеся в модуле POSIX. Чтобы получить сходные строки даты, не зависящие от национальных настроек, присвойте надлежащие значения переменным окружения для национальных настроек (посмотрите, пожалуйста, страницу руководства *perllocale*) и попробуйте:

```
use POSIX qw(strftime);
$now_string = strftime "%a %b %e %H:%M:%S %Y", gmtime;
```

Управляющие последовательности %a и %b, представляющие сокращенные формы дня недели и месяца года, не для всех национальных настроек должны быть трехсимвольными.

goto

```
goto LABEL
goto EXPR
goto &NAME
```

goto *LABEL* находит команду с меткой *LABEL* и продолжает выполнение с нее. Если *LABEL* не найдена, создается исключительная ситуация. Нельзя использовать эту функцию для входа в конструкцию, требующую инициализации, например, подпрограмму или цикл foreach. Нельзя также с ее помощью входить в конструкцию, удаленную в результате оптимизации. Она пригодна для перехода почти в любую точку в динамической области видимости,[1] в том числе из подпрограмм, но для этой цели обычно лучше применять какую-либо другую конструкцию, например last или die. Автор Perl никогда не испытывал потребности в этой форме goto (имеется в виду, в Perl, в C – другое дело).

Достигая еще больших вершин ортогональности (и глубин идиотизма), Perl разрешает goto *EXPR*, что предполагает для *EXPR* иметь значением имя метки, местонахождение которой *гарантированно* не будет разрешено до этапа исполнения, поскольку метка неизвестна на этапе компиляции. Это позволяет применять вычислимые goto, как в FORTRAN, но не является обязательной рекомендацией[2] для стремящихся улучшить возможность сопровождения:

```
goto +("FOO", "BAR", "GLARCH")[$i];
```

Стоящая особняком goto *&NAME* является весьма магической, заменяя вызов указанной подпрограммы на подпрограмму, выполняющуюся в данный момент. Эту конструкцию не стыдно использовать в подпрограммах AUTOLOAD, которым нужно загрузить другую подпрограмму, а затем представить дело так, будто в первую очередь была вызвана эта новая подпрограмма, а не первоначальная (только все изменения @_ в исходной подпрограмме распространяются в заменяющую). После goto даже caller не сможет определить, что первой была вызвана подпрограмма AUTOLOAD.

---

[1] Это значит, что если метка не находится в текущей подпрограмме, осуществляется обратный поиск метки в подпрограммах, вызвавших данную, что делает почти невозможным сопровождение вашей программы.

[2] Это сдержанное высказывание считается забавным, поэтому мы решили попробовать привести его здесь.

grep

```
grep EXPR, LIST
grep BLOCK LIST
```

Эта функция вычисляет *EXPR* или *BLOCK* в булевом контексте для каждого элемента *LIST*, временно устанавливая $_ поочередно в каждый элемент, сходным с foreach образом. В списковом контексте она возвращает список тех элементов, для которых выражение истинно. (Оператор назван в честь любимой программы Unix, извлекающей из файлов строки, соответствующие заданному шаблону. В Perl выражение часто является шаблоном, но не обязано им быть.) В скалярном контексте возвращаемое grep значение показывает, сколько раз выражение было истинным.

Если @all_lines содержит строки кода, то в следующем примере выбрасываются строки комментариев:

```
@code_lines = grep !/^\s*#/, @all_lines;
```

Поскольку $_ представляет собой неявный псевдоним для каждого значения в списке, то изменение $_ вызывает изменение элементов исходного списка. Такое свойство полезно и поддерживается, но иногда приводит к результатам, которые могут оказаться странными и неожиданными. Например:

```
@list = qw(barney fred dino wilma);
@greplist = grep { s/^[bfd]// } @list;
```

@greplist теперь содержит «arney», «red», «ino», но @list содержит «arney», «red», «ino», «wilma»! Программист, Будь Осторожен.

См. также map. Следующие две команды функционально эквивалентны:

```
@out = grep { EXPR } @in;
@out = map { EXPR ? $_ : () } @in
```

hex

```
hex EXPR
hex
```

Эта функция интерпретирует *EXPR* как шестнадцатеричную строку и возвращает эквивалентное десятичное значение. Ведущий 0x, если он есть, игнорируется. Для интерпретации строк, которые могут начинаться с любых из 0, 0b или 0x, используйте oct. В следующем коде $number устанавливается равным 4 294 906 560:

```
$number = hex("ffff12c0");
```

Обратное действие можно выполнить при помощи sprintf:

```
sprintf "%lx", $number;          # (l - буква, а не цифра.)
```

Шестнадцатеричные строки могут представлять только целые числа. Строки, вызывающие переполнение целых, создают исключительную ситуацию.

import

```
import CLASSNAME LIST
import CLASSNAME
```

Встроенной функции `import` нет. Это лишь обычный метод класса, определяемый (или наследуемый) модулями, которые хотят экспортировать имена в другой модуль посредством оператора `use`. Подробности см. в `use`.

index

```
index STR, SUBSTR, OFFSET
index STR, SUBSTR
```

Эта функция ищет одну строку в другой. Она возвращает позицию первого вхождения *SUBSTR* в *STR*. Смещение *OFFSET*, если оно задано, говорит, сколько символов от начала надо пропустить, прежде чем начать поиск. Позиции нумеруются с 0 (или с того, во что вы установили переменную основания индексирования `$[`, – но лучше этого не делать). Если подстрока не найдена, функция возвращает на единицу меньше основания, т. е. обычно `-1`. Чтобы просмотреть всю строку, можно сделать так:

```
$pos = -1;
while (($pos = index($string, $lookfor, $pos)) > -1) {
    print "Found at $pos\n";
    $pos++;
}
```

int

```
int EXPR
int
```

Эта функция возвращает целую часть *EXPR*. Программирующие на C склонны забывать о применении `int` в сочетании с делением, которое является в Perl операцией с плавающей запятой:

```
$average_age = 939/16;      # дает 58.6875 (58 в C)
$average_age = int 939/16;  # дает  58
```

Не следует использовать эту функцию для обычного округления, потому что она производит усечение в сторону 0, а машинное представление чисел с плавающей запятой может иногда приводить к неочевидным результатам. Например, `int(-6.725/0.025)` дает `-268`, а не требуемое `-269`, т. к. деление, в действительности, дает что-то вроде `-268.99999999999994315658`. В большинстве случаев следует предпочесть функции `sprintf`, `printf` или `POSIX::floor` и `POSIX::ceil`, а не `int`.

```
$n = sprintf("%/0f", $f);   # округление (а не усечение) до ближайшего целого
```

ioctl

```
ioctl FILEHANDLE, FUNCTION, SCALAR
```

Эта функция реализует системный вызов *ioctl*(2), который управляет вводом-выводом. Чтобы получить правильные определения функций, вначале, вероятно, придется сказать:

```
require "sys/ioctl.ph";     # возможно, /usr/local/lib/perl/sys/ioctl.ph
```

Если *sys/ioctl.ph* не существует или не содержит правильных определений, придется создать собственные, используя свои заголовочные файлы C, например *sys/ioctl.h*. (Дистрибутив Perl содержит сценарий с именем *h2ph*, который поможет вам это сделать, но запускается он нетривиально.) Со *SCALAR* будет осуществляться чтение или запись (или то и другое) в зависимости от *FUNCTION* — указатель на строковое значение *SCALAR* будет передан в качестве третьего аргумента фактического вызова *ioctl*(2). (Если *SCALAR* не имеет строкового значения, но имеет числовое значение, то будет передано непосредственно это значение, а не указатель на строковое значение.) Для работы со значениями в структурах, используемых `ioctl`, хорошо подходят функции `pack` и `unpack`. В следующем примере с помощью `FIONREAD` `ioctl` определяется, сколько байт доступно для чтения:

```
require 'sys/ioctl.ph';

$size = pack("L", 0);
ioctl(FH, FIONREAD(), $size)
    or die "Couldn't call ioctl: $!\n";
$size = unpack("L", $size);
```

Если *h2ph* не был установлен или не работает, можете поискать ее с помощью *grep* во включаемых файлах вручную или написать маленькую программу C для вывода значения.

Функция `ioctl` (и `fcntl`) возвращает следующие значения:

Значения, возвращаемые системным вызовом	Значения, возвращаемые Perl
`-1`	`undef`
`0`	строка «`0 but true`»
все прочее	это число

Таким образом, Perl возвращает значение «истина» при успехе и «ложь» при неудаче, хотя сохраняется возможность определить фактическое значение, возвращенное операционной системой:

```
$retval = ioctl(...) || -1;
printf " фактически возвратила %d\n", $retval;
```

Специальная строка «`0 but true`» избавляет от предупреждений о недопустимых числовых преобразованиях при использовании ключа `-w`.

Вызовы ioctl нельзя считать переносимыми. Если, скажем, требуется просто отключить эхо для всего сценария, более переносимым способом будет:

```
system "stty -echo";   # Работает в большинстве систем Unix.
```

Если вы *можете* делать что-то в Perl, это не значит, что вы *должны* это делать. Цитируя апостола Павла, «все доступно, но не все благотворно».

Для достижения лучшей переносимости можно воспользоваться модулем Term::ReadKey из CPAN.

join

```
join EXPR, LIST
```

Эта функция соединяет отдельные строки списка *LIST* в одну строку, поля в которой разделяются значением *EXPR*, и возвращает эту строку. Например:

```
$rec = join ':', $login,$passwd,$uid,$gid,$gcos,$home,$shell;
```

Обратное действие осуществляется с помощью функции split. Объединение элементов в поля с фиксированными позициями рассмотрено в описании функции pack. Наиболее эффективный способ конкатенации нескольких строк вместе дает их объединение (join) с нулевой строкой:

```
$string = join "", @array;
```

В отличие от split, join не принимает в качестве первого аргумента шаблон и возбудит исключительную ситуацию, если вы попытаетесь это сделать.

keys

```
keys HASH
```

Эта функция возвращает список, содержащий все ключи заданного хеша. Ключи возвращаются в порядке, представляющемся случайным, но это тот же порядок, который создают функции values и each (в предположении, что хеш не изменялся между вызовами). В качестве побочного эффекта она сбрасывает итератор в *HASH*. Вот (довольно примитивный) способ вывести переменные окружающей среды:

```
@keys   = keys   %ENV;    # ключи имеют тот же порядок, что и
@values = values %ENV;    # значения, как здесь продемонстрировано
while (@keys) {
    print pop(@keys), '=', pop(@values), "\n";
}
```

Более желателен вывод окружения, отсортированного по ключам:

```
foreach $key (sort keys %ENV) {
    print $key, '=', $ENV{$key}, "\n";
}
```

Можно сортировать значения хеша непосредственно, но это довольно бесполезно в отсутствие способа отобразить значения обратно в ключи. Чтобы отсортировать хеш по значениям, как правило, нужно отсортировать ключи, обеспечив функцию сравнения, которая обращается к значениям, основываясь на ключах. Вот пример числовой сортировки хеша в порядке убывания его значений:

```
foreach $key (sort { $hash{$b} <=> $hash{$a} } keys %hash) {
    printf "%4d %s\n", $hash{$key}, $key;
}
```

Использование keys с хешем, связанным с большим файлом DBM, создаст большой список, а в результате вы получите большой процесс. Предпочтительным может оказаться применение в этом случае функции each, которая осуществляет проход через все записи хеша, не создавая из них при этом гигантского списка.

В скалярном контексте keys возвращает число элементов в хеше (и сбрасывает итератор each). Однако чтобы получить эту информацию для связанного хеша, в том числе файлов DBM, Perl должен пройти весь хеш, что не эффективно. В таком случае помогает применение keys в пустом контексте.

При использовании в качестве левого значения keys увеличивает число блоков хеширования, выделенных для данного хеша. (Это похоже на предварительное увеличение размера массива путем присвоения большего значения переменной $#array.) Предварительное расширение хеша может повысить эффективность, если известно, что хеш будет расти, и известно, насколько. Если сказать:

```
keys %hash = 1000;
```

то для %hash будет выделено не менее 1000 блоков (фактически вы получите 1024 блока, т. к. округление происходит до очередной степени двух). Сократить количество блоков, выделенных для хеша, используя keys указанным способом, нельзя (но если – случайно – попытаться сделать это, ничего не произойдет). Блоки сохраняются, даже если выполнить %hash = (). Обратитесь к undef %hash, если требуется освободить память, когда %hash все еще находится в области видимости.

См. также each, values и sort.

kill

```
kill SIGNAL, LIST
```

Эта функция посылает сигнал списку процессов. В качестве *SIGNAL* может выступать целое число или имя сигнала в кавычках (без "SIG" впереди). Попытка задать нераспознаваемое имя сигнала влечет вызов исключительной ситуации. Функция возвращает число процессов, которым успешно передан сигнал. Если *SIGNAL* является отрицательным числом, функция убивает не отдельные процессы, а их группы. (В SysV отрицательный номер процесса

тоже убивает группы процессов, но такой способ не переносим.) PID, равный нулю, посылает сигнал всем процессам с таким же ID группы, как у отправителя. Например:

```
$cnt = kill 1, $child1, $child2;
kill 9, @goners;
kill 'STOP', getppid       # Можно *так* приостановить свою оболочку регистрации...
        unless getppid == 1;    # (Но не дразните init(8).)
```

Сигнал *SIGNAL*, равный 0, проверяет, жив ли процесс и можно ли еще передать ему сигнал. Сигнал не посылается. Таким способом можно удостовериться, что процесс еще существует и не сменил свой UID.

```
use Errno qw(ESRCH EPERM);
if (kill 0 => $minion) {
    print "$minion is alive!\n";
} elsif ($! == EPERM) {             # изменил UID
    print "$minion вышел из-под контроля!\n";
} elsif ($! == ESRCH) {
    print "$minion скончался.\n";  # или  стал зомби
} else {
    warn "Странно, не удалось проверить статус $minion: $!\n";
}
```

См. раздел «Сигналы» главы 16.

last

```
last LABEL
last
```

Оператор last осуществляет немедленный выход из данного цикла, как команда break в C или Java (при использовании в циклах). Если метка *LABEL* опущена, оператор относится к самому внутреннему охватывающему циклу. Блок continue, если он есть, не выполняется.

```
LINE: while (<MAILMSG>) {
    last LINE if /^$/; # выйти после конца заголовка
    # здесь остальная часть цикла
}
```

last нельзя применять для выхода из блока, который возвращает значение, например, eval {}, sub {} или do {}, не должен он использоваться и для выхода из операции grep или map. Если включен вывод предупредительных сообщений, Perl предупреждает о применении last для выхода из цикла, который находится вне текущей лексической области видимости, например, цикла в вызывающей подпрограмме.

Собственно блок семантически идентичен циклу, выполняемому один раз. Поэтому last можно использовать для обеспечения досрочного выхода из такого блока.

См. также в главе 4 примеры, показывающие работу last, next, redo и continue.

lc

$_ T

```
lc EXPR
lc
```

Эта функция возвращает версию *EXPR*, в которой все знаки находятся на нижнем регистре. Это внутренняя строка, которая реализует escape-последовательность `\L` в строках, заключенных в двойные кавычки. Текущая установка `LC_CTYPE` учитывается, если действует `use locale`, хотя взаимодействие национальных установок с Unicode является, как говорят, предметом текущих исследований. Последние результаты можно найти на странице руководства *perllocale*.

lcfirst

$_ T

```
lcfirst EXPR
lcfirst
```

Функция возвращает версию *EXPR*, первый символ которой находится на нижнем регистре. Это внутренняя строка, которая реализует escape-последовательность `\l` в строках, заключенных в двойные кавычки. Мы учитываем текущую установку `LC_CTYPE` посредством включения директивы `use locale`; надо также понять, как это действует в Unicode.

length

$_

```
length EXPR
length
```

Эта функция возвращает длину в символах скалярной величины *EXPR*. Если *EXPR* опущено, она возвращает длину `$_`. (Однако смотрите, чтобы дальнейший текст не выглядел как начало *EXPR*, иначе лексический анализатор Perl собьется. Например, `length < 10` не будет компилироваться. При наличии сомнений используйте круглые скобки.)

Не пытайтесь применять `length` для определения размера массива или хеша. Для массива следует предпочесть `scalar @array`, а для числа пар ключ/значение в хеше — `scalar keys %hash`. (`scalar` обычно опускается, если является избыточным.)

Найти длину строки в байтах, а не в символах, можно так:

```
$blen = do { use bytes; length $string; };
```

или:

```
$blen = bytes::length($string);    # вначале нужно use bytes
```

link

$! X U X T

```
link OLDFILE, NEWFILE
```

Функция создает новое имя файла, являющееся ссылкой на старое имя файла. Возвращает значение «истина» в случае успеха и «ложь» в противном случае. См. также `symlink` далее в этой главе. Маловероятно, что эта функция окажется реализованной в системах, отличающихся от Unix.

listen

```
listen SOCKET, QUEUESIZE
```

Сообщает системе, что принимать соединения предполагается на данном сокете *SOCKET* и что система может помещать ожидающие соединения в очередь размером, определяемым *QUEUESIZE*. Представьте себе, что на телефоне есть очередь ожидающих вызовов, в которую могут встать до 17 позвонивших. (Руки-ноги трясутся!) Функция возвращает значение «истина» в случае успеха и «ложь» в противном случае.

```
use Socket;
listen(PROTOSOCK, SOMAXCONN)
    or die "невозможно установить очередь ожидания на PROTOSOCK: $!";
```

См. описание функции `accept`. См. также раздел «Сокеты» главы 16 и описание *listen*(2).

local

```
local EXPR
```

Этот оператор не создает локальную переменную, для этой цели применяется `my`. Вместо этого он локализует существующие переменные, т. е. в результате одна или более переменных получают значения с локальной областью видимости в самом внутреннем охватывающем блоке, `eval` или файле. Если перечислено несколько переменных, список должен быть помещен в круглые скобки, поскольку этот оператор связывает более сильно, чем запятые. Все перечисленные переменные должны быть законными левыми значениями, т. е. допускать присвоение им значений. В их число могут входить отдельные элементы массивов и хешей.

При выполнении этого оператора текущие значения указанных переменных сохраняются в скрытом стеке и восстанавливаются после выхода из блока, подпрограммы, `eval` или файла. После выполнения `local`, но перед выходом из области видимости, все подпрограммы и выполняемые форматы будут видеть локальное (внутреннее) значение, а не предыдущее (внешнее), поскольку переменная остается глобальной, несмотря на наличие у нее локального значения. Техническим термином для этого служит «динамическая область видимости». См. раздел «Объявления с областью видимости» главы 4.

Выражению *EXPR* можно при желании присваивать значение, что позволяет инициализировать переменные при их локализации. Если инициализатора нет, все переменные получают значение `undef`, а все массивы и хеши – значение `()`. Как и при обычном присвоении, если переменные заключены в круг-

лые скобки (или если переменная является массивом или хешем), выражение справа вычисляется в списковом контексте. В противном случае выражение справа вычисляется в скалярном контексте.

В любом случае выражение справа вычисляется перед локализацией, но инициализация происходит после локализации, поэтому можно инициализировать локализованную переменную ее нелокализованным значением. К примеру, следующий код показывает, как выполнить временную модификацию глобального массива:

```
if ($sw eq '-v') {
    # инициализировать локальный массив глобальным
    local @ARGV = @ARGV;
    unshift @ARGV, 'echo';
    system @ARGV;
}
# @ARGV восстановлен
```

Можно также временно модифицировать глобальные хеши:

```
# временно добавить пару записей к хешу %digits
if ($base12) {
    # (ЗАМЕЧАНИЕ: Мы не утверждаем, что это эффективно!)
    local(%digits) = (%digits, T => 10, E => 11);
    parse_num();
}
```

Посредством `local` можно присвоить временные значения отдельным элементам массивов и хешей, даже имеющим лексическую область видимости:

```
if ($protected) {
    local $SIG{INT} = 'IGNORE';
    precious();     # никаких прерываний во время этой функции
}                   # предыдущий обработчик (если он был) восстановлен
```

Можно также использовать `local` с переменными типа typeglob для создания локальных указателей файлов без загрузки объемистых модулей объектов:

```
local *MOTD;                # защита глобальных указателей MOTD
my $fh = do { local *FH };  # создать новый косвенный указатель
```

(В версии Pedrl 5.6 достаточно простого `my $fh;`, поскольку если передать неопределенную переменную туда, где ожидается реальный указатель файла, например, первым аргументом `open` или `socket`, Perl «самооживит» для вас новенький указатель файла.)

Но в целом обычно следует использовать `my`, а не `local`, поскольку большинство людей на практике не представляют себе `local` как «локальный». См. описание `my`.

localtime

```
localtime EXPR
localtime
```

Функция преобразует значение, возвращаемое time, в список из девяти элементов, представляющих время, приведенное к местному часовому поясу. Обычно она используется так:

```
# 0    1    2     3     4    5     6     7     8
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = localtime;
```

Если, как в данном случае, *EXPR* опущено, выполняется localtime(time()).

Все элементы списка представляют собой числа и поступают прямо из struct tm. (Это жаргон программирования на C – пусть он вас не беспокоит.) В частности, это означает, что $mon имеет диапазон 0..11, при этом январь имеет номер 0, а $wday имеет диапазон 0..6, при этом воскресенье имеет номер 0. Можно запомнить, какие величины начинают нумероваться с нуля, поскольку это те, которые всегда используются в качестве индексов в нумерующихся с нуля массивах, содержащих названия месяцев и дней недели.

Например, получить название текущего дня недели можно так:

```
$thisday = (Sun,Mon,Tue,Wed,Thu,Fri,Sat)[(localtime)[6]];
```

$year – количество лет, прошедших после 1900; таким образом, в 2023 году $year будет иметь значение 123, а *не* просто 23. Чтобы получить 4-значный год, просто скажите $year + 1900. Чтобы получить 2-значный год (например, «01» в 2001 году), используйте sprintf("%02d", $year % 100).

Модуль библиотеки Perl Time::Local содержит подпрограмму timelocal, которая может осуществлять преобразование в обратном направлении.

В скалярном контексте localtime возвращает строку в стиле *ctime*(3). Например, команду *date*(1) можно (почти)[1] эмулировать с помощью:

```
perl -le 'print scalar localtime'
```

См. также функцию strftime в стандартном модуле POSIX, реализующей более развитые возможности форматирования времени. Модуль Time::localtime поддерживает интерфейс к этой функции через имена.

lock

```
lock THING
```

Функция lock помещает блокировку на переменную, подпрограмму или объект, на который ссылается *THING*, пока блокировка не выйдет из области видимости. Для обратной совместимости эта функция реализуется как встро-

[1] *date*(1) выводит часовой пояс, а localtime – нет.

енная, но только если ваша версия Perl компилировалась с включением поддержки потоков и вы сказали use Threads. В противном случае Perl воспринимает ее как функцию, определяемую пользователем. См. главу 17 «Потоки».

log

$_ $@

```
log EXPR
log
```

Эта функция возвращает натуральный логарифм (т. е. по основанию *e*) *EXPR*. Если *EXPR* меньше нуля, создается исключительная ситуация. Для получения логарифма по другому основанию используйте элементарную алгебру: логарифм числа по основанию *N* равен натуральному логарифму этого числа, деленному на натуральный логарифм *N*. Например:

```
sub log10 {
    my $n = shift;
    return log($n)/log(10);
}
```

Обратной для log функцией является exp. См. exp.

lstat

$_ $! X U

```
lstat EXPR
lstat
```

Эта функция делает то же, что функция Perl stat (включая установку специального указателя файла _), но если последняя часть имени файла представляет собой символическую ссылку, выполняется stat с самой символической ссылкой, а не файлом, на который она указывает. (Если символические ссылки не реализованы в системе, выполняется обычная stat.)

m//

T X T

```
/PATTERN/
m/PATTERN/
```

Это оператор поиска по шаблону, который интерпретирует *PATTERN* как регулярное выражение. Оператор анализируется как строка в двойных кавычках, а не функция. См. главу 5 «Поиск по шаблону».

map

```
map BLOCK LIST
map EXPR, LIST
```

Эта функция вычисляет *BLOCK* или *EXPR* для каждого элемента списка *LIST* (локально устанавливая переменную $_ равной каждому элементу) и возвращает список, объединяющий результаты всех таких вычислений. Она вычис-

ляет *BLOCK* или *EXPR* в списковом контексте, поэтому каждый элемент *LIST* может отображаться в ноль, один или более элементов в возвращаемом значении. Все они уплощаются в один список. Например:

```
@words = map { split ' ' } @lines;
```

расщепляет список строк в список слов. Но часто существует взаимно однозначное соответствие между входными и выходными значениями:

```
@chars = map chr, @nums;
```

транслирует список чисел в список соответствующих символов. А вот пример отображения один к двум:

```
%hash = map { genkey($_) => $_ } @array;
```

что является лишь своеобычным вариантом следующей записи:

```
%hash = ();
foreach $_ (@array) {
    $hash{genkey($_)} = $_;
}
```

Поскольку $_ представляет собой псевдоним (неявную ссылку) значения списка, эту переменную можно использовать для изменения элементов массива. Это удобно и поддерживается, хотя может порождать странные результаты, когда *LIST* не является именованным массивом. Применение с этой целью обычного цикла foreach может оказаться более ясным. См. также grep; map отличается от grep тем, что map возвращает список, состоящий из результатов всех последовательных вычислений *EXPR*, тогда как grep возвращает список, состоящий из всех значений *LIST*, для которых *EXPR* имеет значение «истина».

mkdir

```
mkdir FILENAME, MASK
mkdir FILENAME
```

Эта функция создает каталог, заданный аргументом *FILENAME*, инициализируя для него права доступа, заданные числовой маской *MASK* с модификацией текущей umask. В случае успеха операции она возвращает значение «истина», в противном случае – значение «ложь».

Если параметр *MASK* опущен, используется маска 0777, и это почти всегда то, что требуется. В целом создавать каталоги с разрешительной маской *MASK* (как 0777) и разрешать пользователю модифицировать ее с помощью своей umask лучше, чем задавать ограничительную маску и не дать пользователю возможность быть более снисходительным. Исключение составляют каталоги и файлы, которые должны быть закрытыми (например, почтовые файлы). См. umask.

Если системный вызов *mkdir*(2) не встроен в вашу библиотеку C, Perl эмулирует его, вызывая программу *mkdir*(1) для каждого каталога. При необходимости создания большого списка каталогов в такой системе более эффективно самостоятельно вызвать программу *mkdir* со списком каталогов, чем запускать тьму подпроцессов.

msgctl

$! X U

```
msgctl ID, CMD, ARG
```

Эта функция выполняет системный вызов System V IPC *msgctl*(2); подробности см. в *msgctl*(2). Может потребоваться сначала выполнить `use IPC::SysV`, чтобы получить правильные определения констант. Если *CMD* имеет значение `IPC_STAT`, то *ARG* должен быть переменной, в которой будет находиться возвращаемая структура C `msqid_ds`. Возвращаемые значения – такие же, как в `ioctl` и `fcntl`: `undef` для ошибки, `"0 but true"` для нуля и фактическое возвращаемое значение в остальных случаях.

Эта функция доступна только на машинах, поддерживающих System V IPC, которых значительно меньше, чем поддерживающих сокеты.

msgget

$! X U

```
msgget KEY, FLAGS
```

Эта функция выполняет системный вызов System V IPC *msgget*(2). Подробности см. в *msgget*(2). Функция возвращает ID очереди сообщений или `undef` в случае возникновения ошибки. Перед вызовом необходимо выполнить `use IPC::SysV`.

Эта функция доступна только на машинах, поддерживающих System V IPC.

msgrcv

$! X U

```
msgrcv ID, VAR, SIZE, TYPE, FLAGS
```

Функция выполняет системный вызов *msgrcv*(2) для получения сообщения из очереди сообщения *ID* в переменную *VAR* при максимальном размере сообщения *SIZE*. Подробности см. в *msgrcv*(2). При получении сообщения в начале *VAR* находится тип сообщения, а максимальная длина *VAR* вычисляется как *SIZE* плюс размер типа сообщения. Функция возвращает значение «истина» в случае успеха и «ложь» при возникновении ошибки. Перед вызовом необходимо выполнить `use IPC::SysV`.

Эта функция доступна только на машинах, поддерживающих System V IPC.

msgsnd

$! X U

```
msgsnd ID, MSG, FLAGS
```

Эта функция вызывает системный вызов *msgsnd*(2) для отправки сообщения *MSG* в очередь сообщений *ID*. Подробности см. в *msgsnd*(2). *MSG* должно начи-

наться с длинного целого, представляющего тип сообщения. Создать сообщение можно так:

```
$msg = pack "L a*", $type, $text_of_message;
```

Функция возвращает значение «истина» в случае успеха и «ложь» при возникновении ошибки. Перед вызовом необходимо выполнить `use IPC::SysV`.

Эта функция доступна только на машинах, поддерживающих System V IPC.

my

```
my TYPE EXPR : ATTRIBUTES
my EXPR : ATTRIBUTES
my TYPE EXPR
my EXPR
```

Этот оператор объявляет одну или более закрытых переменных, которые будут существовать только в самом внутреннем охватывающем блоке, подпрограмме, `eval` или файле. Если перечислено несколько переменных, список должен быть помещен в круглые скобки, поскольку этот оператор связывает более сильно, чем запятые. Объявлять таким способом можно только простые скаляры или целые массивы и хеши.

Имя переменной не может квалифицироваться именем пакета, поскольку все переменные пакета глобально доступны через соответствующие им таблицы символов, а лексические переменные не связаны с какими-либо таблицами символов. Поэтому, в отличие от `local`, этот оператор не имеет к глобальным переменным никакого отношения, за исключением того, что он маскирует все переменные с такими же именами в своей области видимости (т. е. там, где существуют закрытые переменные). Однако доступ к глобальной переменной всегда может быть осуществлен через квалификацию именем пакета или через символическую ссылку.

Область видимости закрытой переменной начинается лишь с команды, находящейся *после* ее объявления. Область видимости переменной простирается на все последующие вложенные блоки, вплоть до конца самой области видимости.

Однако это означает, что любые подпрограммы, которые вызываются из области видимости закрытой переменной, не могут видеть эту закрытую переменную, если только блок, содержащий определение самой подпрограммы, не является текстуально вложенным в область видимости этой переменной. Это звучит сложно, пока вы с этим не разберетесь. Техническим термином для этого служит лексическая *область видимости* (*lexical scoping*), поэтому такие переменные мы часто называем *лексическими*. В культуре C их часто называют «автоматическими» переменными, поскольку они автоматически размещаются и удаляются при входе и выходе из области видимости.

Выражению *EXPR* можно при желании присваивать значение, что позволяет инициализировать лексические переменные. (Если инициализатора нет, все

скаляры получают неопределенное значение, а все массивы и хеши – пустой список). Как и при обычном присвоении, если переменные слева заключаются в круглые скобки (или если переменная является массивом или хешем), выражение справа вычисляется в списковом контексте. В противном случае выражение справа вычисляется в скалярном контексте. Например, можно назвать формальные параметры подпрограммы с помощью присвоения списка:

```
my ($friends, $romans, $countrymen) = @_;
```

Но следите за тем, чтобы не опустить скобки, указывающие на присвоение списка, например:

```
my $country = @_;  # правильно или нет?
```

В этом коде переменной присваивается длина массива (т. е. число аргументов подпрограммы), поскольку массив вычисляется в скалярном контексте. Однако можно с пользой применить скалярное присвоение формальному параметру, если обратиться к оператору `shift`. На практике, поскольку методам объектов в качестве первого аргумента передается объект, многие подпрограммы методов начинают с того, что «крадут» первый аргумент:

```
sub simple_as {
    my $self = shift;   # присвоение скаляра
    my ($a,$b,$c) = @_; # присвоение списка
    ...
}
```

Если попытаться объявить подпрограмму с лексической областью видимости как `my sub`, Perl умрет с сообщением, что эта функция пока не реализована. (Если, конечно, она *уже* не реализована.)

Параметры *TYPE* и *ATTRIBUTES* являются необязательными, что и неплохо, поскольку оба считаются экспериментальными. Вот как могло бы выглядеть использующее их объявление:

```
my Dog $spot :ears(short) :tail(long);
```

TYPE, если он задан, указывает, какого типа скаляр или скаляры объявлены в *EXPR*, либо непосредственно как одна или более скалярных переменных, либо косвенно через массив или хеш. Если *TYPE* является именем класса, то предполагается, что скаляры содержат ссылки на объекты этого типа или объекты, совместимые с этим типом. В частности, совместимыми считаются производные классы. Следовательно, предполагая, что `Collie` происходит от `Dog`, можно объявить:

```
my Dog $lassie = new Collie;
```

Данное объявление сообщает, что объект `$lassie` будет использоваться способом, совместимым с тем, что это объект `Dog`. То обстоятельство, что фактически это объект `Collie`, не должно иметь значения до тех пор, пока с ним де-

лают то, что можно делать с Dog. Благодаря магии виртуальных методов реализация этих методов Dog вполне может находиться в классе Collie, но вышеприведенное объявление говорит только об интерфейсе, не о реализации. Теоретически.

Интересно, что вплоть до версии 5.6.0 единственный случай, когда Perl обращает внимание на объявление *TYPE*, это когда соответствующий класс объявил поля с помощью директивы use fields. Совместно эти объявления позволяют коду, находящемуся вне класса, «видеть» реализацию класса в виде псевдохеша, поэтому поиск в псевдохеше может быть оптимизирован компилятором в поиск в массиве. В некотором смысле псевдохеш *является* интерфейсом к такому классу, поэтому наша теория остается в силе, даже потерпев незначительный урон. Подробнее о псевдохешах см. раздел «Псевдохеши» главы 8 «Ссылки».

В будущем другие типы классов могут интерпретировать *TYPE* иным образом. Объявление *TYPE* следует рассматривать как родовой интерфейс, который когда-нибудь будет реализовываться различными способами в зависимости от класса. На самом деле *TYPE* может даже не быть официальным именем класса. Мы зарезервировали имена типов на нижнем регистре для Perl, потому что один из способов, которыми мы хотели бы расширить интерфейс типа, состоит в разрешении необязательных объявлений типов низкого уровня, таких как int, num, str и ref. Эти объявления не будут служить строгой типизации; они будут служить советами компилятору в отношении того, как оптимизировать хранение переменной, предполагая, что переменная по большей части будет использоваться так, как она объявлена. Семантика скаляров останется, в основном, той же – по-прежнему можно будет складывать два скаляра str или выводить скаляр int, как если бы они были обычными полиморфными скалярами, с которыми вы знакомы. Но для переменной, объявленной как int, Perl мог бы хранить только целое значение и забыть о кэшировании результирующей строки, которое он осуществляет сейчас. Циклы с переменной цикла int могли бы работать быстрее, особенно в коде, скомпилированном в C. В частности, числовые массивы можно было бы хранить гораздо более компактным способом. В предельном случае встроенная функция vec может стать устаревшей, если будет позволено писать такие объявления:

```
my bit @bitstring;
```

Объявление *ATTRIBUTES* – еще более экспериментальное. Мы не много сделали сверх того, чтобы зарезервировать синтаксис и создать прототип внутреннего интерфейса; подробнее об этом см. описание директивы use attributes в главе 31. Первым атрибутом, который мы реализуем, будет, вероятно, constant:

```
my num $PI : constant = atan2(1,1) * 4;
```

Но есть много других способов, например, можно установить значения по умолчанию для массивов и хешей или позволить переменным совместно

использоваться несколькими интерпретаторами. Подобно интерфейсу типа, интерфейс атрибутов должен рассматриваться как родовой интерфейс, своего рода рабочее место для изобретения новых синтаксиса и семантики. Мы не знаем, как будет развиваться Perl в течение следующих 10 лет. Мы знаем только, что можем облегчить себе жизнь, планируя заранее это развитие.

См. также `local`, `our` и раздел «Объявления с областью видимости» главы 4.

new

```
new CLASSNAME LIST
new CLASSNAME
```

Не существует встроенная функция `new`. Это просто обычный метод конструктора (т. е. подпрограмма, определяемая пользователем), которая определена или наследуется классом *CLASSNAME* (т. е. пакетом), чтобы разрешить создание объектов типа *CLASSNAME*. Многие конструкции называются «new», но только по договоренности, лишь бы обмануть программистов C++, чтобы им казалось, что они понимают происходящее. Всегда читайте документацию к классу, чтобы знать, как вызывать его конструкторы; например, конструктор, создающий окно списка в наборе примитивов Tk, называется просто `Listbox()`. См. главу 12.

next

```
next LABEL
next
```

Оператор `next` аналогичен команде `continue` в C: он начинает очередной проход цикла, обозначенного меткой *LABEL*:

```
LINE: while (<STDIN>) {
    next LINE if /^#/;     # отбросить комментарии
    ...
}
```

Если бы в этом примере был блок `continue`, он был бы выполнен сразу после вызова `next`. Если *LABEL* опущена, то оператор ссылается на самый внутренний охватывающий цикл.

Собственно блок семантически идентичен циклу, выполняемому один раз. Поэтому `next` осуществляет досрочный выход из такого блока (через блок `continue`, если он есть).

Нельзя организовывать при промощи `next` выход из блока, который возвращает значение, например `eval {}`, `sub {}` или `do {}`, кроме того, он не должен применяться для выхода из операции `grep` или `map`. Если включен вывод предупредительных сообщений, Perl предупреждает, когда `next` используется для выхода из цикла, который находится вне текущей лексической области видимости, например, цикла в вызывающей подпрограмме. См. раздел «Команды цикла» в главе 4.

no

$@

```
no MODULE LIST
```

Ознакомьтесь с описанием оператора use, который представляет собой своего рода противоположность no. Большинство стандартных модулей не осуществляет обратного импорта чего-либо, что, в сущности, делает no пустой операцией. Модули прагм склонны быть здесь более любезными. Если *MODULE* не удается найти, возбуждается исключительная ситуация.

oct

$_

```
oct EXPR
oct
```

Эта функция интерпретирует *EXPR* как восьмеричную строку и возвращает эквивалентное десятичное значение. Если строка *EXPR* начинается с "0x", она интерпретируется как шестнадцатеричная, а если с "0b", то как строка двоичных знаков. Следующий код правильно преобразует любые входные строки в десятичном, двоичном, восьмеричном и шестнадцатеричном основании, записанные в стандартной нотации C или C++:

```
$val = oct $val if $val =~ /^0/;
```

Обратное преобразование можно выполнить с помощью sprintf с соответствующим форматом:

```
$perms = (stat("filename"))[2] & 07777;
$oct_perms = sprintf "%lo", $perms;
```

Функция oct обычно используется, когда строка данных, такая как "644", должна быть преобразована, например, в режим файла. Хотя Perl автоматически преобразует строки в числа по мере надобности, такое автоматическое преобразование предполагает использование основания 10.

open

$! X ARG X U X T

```
open FILEHANDLE, MODE, LIST
open FILEHANDLE, EXPR
open FILEHANDLE
```

Функция open связывает внутренний указатель файла *FILEHANDLE* со спецификацией внешнего файла, заданной *EXPR* или *LIST*. Ее можно вызывать с одним, двумя или тремя аргументами (или более, если третий аргумент является командой и вы выполняете версию Perl не ниже 5.6.1). Если имеется три или более аргументов, второй аргумент указывает режим доступа *MODE*, в котором должен быть открыт файл, а третий аргумент (*LIST*) предоставляет фактические имя файла или команду для исполнения, в зависимости от режима. В случае команды могут быть переданы дополнительные аргументы, если вы хотите вызвать команду, например, system или exec, непосредственно,

без привлечения оболочки. Либо команда может быть передана как один аргумент (третий), и в таком случае решение о вызове оболочки зависит от того, содержит ли команда метасимволы. (Не указывайте более трех аргументов, если аргументы представляют собой обычные имена файлов: это не сработает.) Если *MODE* не распознан, open возбуждает исключительную ситуацию.

Если присутствуют только два аргумента, предполагается, что режим и имя файла/команда объединены во втором аргументе. (А если не указать во втором аргументе режим, а только имя файла, тогда файл, по соображениям осторожности, открывается только для чтения.)

Если аргумент только один, то скалярная переменная пакета с таким же именем, как *FILEHANDLE*, должна содержать имя файла и необязательный режим:

```
$LOG = ">logfile";        # $LOG не должна объявляться как my!
   open LOG or die "Невозможно открыть logfile: $!";
```

Но так делать не надо. Это плохой стиль. Забудьте, что вы это видели.

Функция open возвращает значение «истина» в случае успеха и undef в противном случае. Если open открывает канал в дочерний процесс, возвращаемым значением будет ID процесса этого нового процесса. Как и при любом системном вызове, всегда проверяйте значение, возвращаемое функцией open, чтобы убедиться в ее выполнении. Но это не C или Java, поэтому избегайте применения команды if, если можно ограничиться оператором or. Можно также использовать ||, но в этом случае укажите в вызове open круглые скобки. Решив опустить скобки в вызове функции, чтобы превратить ее в оператор списка, следите за тем, чтобы после списка применялось «or die», а не «|| die», потому что приоритет || выше, чем списковых операторов типа open, а || свяжет ваш последний аргумент, а не open в целом:

```
open LOG, ">logfile" || die "Невозможно создать logfile: $!";  # НЕВЕРНО
open LOG, ">logfile" or die "Невозможно создать logfile: $!";  # ok
```

Это выглядит довольно напряженно, но обычно вводят какие-то пробельные символы, чтобы было видно, где заканчивается оператор списка:

```
open LOG, ">logfile"
        or die "Невозможно создать logfile: $!";
```

Как показывает этот пример, аргумент *FILEHANDLE* часто является простым идентификатором (обычно в верхнем регистре), но может также представлять собой выражение, значение которого дает ссылку на фактический указатель файла. (Ссылка может быть символической ссылкой на имя указателя файла или жесткой ссылкой на любой объект, который может быть интерпретирован как указатель файла.) Это называется *косвенным указателем файла* (*indirect filehandle*), и любая функция, принимающая *FILEHANDLE* в качестве первого аргумента, может обрабатывать косвенные указатели файлов так же, как прямые. Но open имеет такую особенность – если передать ей в качестве косвенного указателя файла неопределенную переменную, то Perl автоматически определит эту переменную, т. е. самооживит ее, поместив в нее ссылку на надлежащий указатель файла. Одно из вытекающих

преимуществ состоит в том, что указатель файла будет автоматически закрыт, когда не останется действующих ссылок на него, – обычно, когда переменная выходит из области видимости:

```
{
        my $fh;                     # (не инициализирована)
        open($fh, ">logfile")       # $fh самооживлена
            or die "Невозможно создать logfile: $!";
        ...                         # сделать что-то с $fh
}                                   # $fh здесь закрывается
```

Объявление `my $fh` можно хорошо читаемым образом включить в `open`:

```
open my $fh, ">logfile" or die ...
```

Символ >, который вы видели перед именем файла, является примером режима. Исторически вначале появилась форма `open` с двумя аргументами. Недавнее добавление трехаргументной формы позволяет отделить режим от имени файла и избежать смешения обоих. В следующем примере предполагается, что пользователь не пытается открыть имя файла, которое начинается с «>». Мы можем быть уверены, что задается режим «>», который открывает файл, поименованный в *EXPR*, для записи создается файл, если он не существует, и файл усекается до нуля, если он уже существует:

```
open(LOG, ">", "logfile")  or die "Can't create logfile: $!";
```

В более коротких формах имя файла и режим находятся в одной строке. Строка анализируется во многом так же, как обычная оболочка обрабатывает переадресацию файлов и каналов. Во-первых, из строки удаляются ведущие и замыкающие пробельные символы. Затем строка исследуется с обоих концов, если нужно, на наличие символов, указывающих, как должен быть открыт файл. Между режимом и именем файла допускается наличие пробельного символа.

Режимы, указывающие на то, как должен открываться файл, являются символами переадресации в стиле оболочки. Список этих символов приведен в табл. 29.1. (Для доступа к файлу с комбинацией режимов открытия, не входящей в данную таблицу, обращайтесь к функции низкого уровня `sysopen`.)

Таблица 29.1. Режимы для open

Режим	Доступ по чтению	Доступ по записи	Только для дополнения	Создать несуществующий	Разрушить существующий
< PATH	Да	Нет	Нет	Нет	Нет
> PATH	Нет	Да	Нет	Да	Да
>> PATH	Нет	Да	Да	Да	Нет
+< PATH	Да	Да	Нет	Нет	Нет
+> PATH	Да	Да	Нет	Да	Да
+>> PATH	Да	Да	Да	Да	Нет
| COMMAND	Нет	Да	n/a	n/a	n/a
COMMAND |	Да	Нет	n/a	n/a	n/a

Если установлен режим «<» или никакой не установлен, то существующий файл открывается для ввода. В режиме «>» файл открывается для вывода, что усекает существующие файлы и создает несуществующие, а в режиме «>>» файл создается при необходимости и открывается для дописывания (при этом весь вывод автоматически помещается в конец файла. Если новый файл создается при использовании режима «>» или «>>», а файл ранее не существовал, права доступа зависят от текущей umask процесса согласно правилам, описываемым для этой функции.

Вот типичные примеры:

```
open(INFO,      "datafile")  || die("невозможно открыть datafile: $!");
open(INFO,    "< datafile")  || die("невозможно открыть datafile: $!");
open(RESULTS, "> runstats")  || die("невозможно открыть runstats: $!");
open(LOG,    ">> logfile ")  || die("невозможно открыть logfile:  $!");
```

Те, кто предпочитает использовать меньше знаков пунктуации, могут написать:

```
open INFO,      "datafile"   or die "невозможно открыть datafile: $!";
open INFO,    "< datafile"   or die "невозможно открыть datafile: $!";
open RESULTS, "> runstats"   or die "невозможно открыть runstats: $!";
open LOG,    ">> logfile "   or die "невозможно открыть logfile:  $!";
```

При открытии для чтения специальное имя файла "-" относится к STDIN. При открытии для записи то же специальное имя файла относится к STDOUT. Обычно они задаются как «<-» и «>-» соответственно.

```
open(INPUT,  "-" ) or die;     # повторно открыть стандартный ввод для чтения
open(INPUT,  "<-") or die;     # то же, но явно
open(OUTPUT, ">-") or die;     # повторно открыть стандартный вывод для записи
```

Благодаря этому можно передать в программу имя файла, который будет использовать стандартный ввод или вывод, но автору программы не надо писать специальный код, чтобы узнать это.

Можно также поместить «+» перед любым из этих трех режимов, чтобы запросить одновременно чтение и запись. Тем не менее, знаки «меньше» и «больше» по-прежнему определяют, будет ли файл усечен или создан, а также должен ли он уже существовать. Это означает, что «+<» почти всегда предпочтительнее для чтения/обновления по записи, а сомнительный режим «+>» разрушит содержимое файла, прежде чем можно будет что-нибудь из него прочесть. (Устанавливайте этот режим, только если необходимо заново прочесть то, что было только что записано.)

```
open(DBASE, "+< database")
    or die "невозможно открыть существующую БД в режиме обновления: $!";
```

Можно рассматривать файл, открытый в режиме обновления, как базу данных с произвольным доступом и перемещаться к байту с заданным номером при помощи seek, но переменная длина записей в обычных текстовых файлах обычно делает непрактичным использование режима чтения-записи для

обновления таких файлов. О другом подходе к обновлению говорится в описании параметра командной строки -i в главе 19.

Если ведущий символ *EXPR* является символом канала (конвейера), open создает новый процесс и соединяет с командой указатель, открытый только для записи. В результате можно осуществлять запись по этому указателю, и то, что мы записываем, будет попадать на стандартный ввод команды. Например:

```
open(PRINTER, "| lpr -Plp1")    or die "невозможно  выполнить: $!";
print PRINTER "stuff\n";
close(PRINTER)                  or die "неуспех lpr/close: $?/$!";
```

Если замыкающий символ *EXPR* представляет собой символ канала, то open снова запускает новый процесс, но на этот раз с ним соединяется указатель файла только для чтения. Благодаря этому вывод команды на STDOUT появляется в вашем указателе для чтения. Например:

```
open(NET, "netstat -i -n |")    or die "невозможно выполнить : $!";
while (<NET>) { ... }
close(NET)                      or die "невозможно закрыть netstat: $!/$?";
```

В результате явного закрытия указателя файла для канала родительский процесс ждет, когда дочерний процесс завершит работу, и возвращает код статуса в $? ($CHILD_ERROR). Возможно также, что close установит $! ($OS_ERROR). См. в примерах для close и system, как интерпретировать эти коды ошибок.

Любая команда канала, содержащая метасимволы оболочки, например, групповые символы или переадресацию ввода-вывода, передается канонической оболочке системы (*/bin/sh* в Unix), чтобы сначала были обработаны эти специфические для оболочки конструкции. Если метасимволы не обнаружены, Perl запускает новый процесс сам, не вызывая оболочку.

Для запуска каналов можно использовать и формат с тремя аргументами. Эквивалентом предшествующего открытия канала в таком стиле будет:

```
open(PRINTER, "|-", "lpr -Plp1")    or die "невозможно выполнить : $!";
open(NET, "-|", "netstat -i -n")    or die "невозможно выполнить : $!";
```

Здесь минус во втором аргументе представляет команду в третьем аргументе. При этих командах не выполняется вызов оболочки, но если необходимо гарантировать отсутствие обработки оболочкой, то (в новых версиях Perl) можно сказать:

```
open(PRINTER, "|-", "lpr", "-Plp1")    or die "невозможно выполнить: $!";
open(NET, "-|", "netstat", "-i", "-n") or die "невозможно выполнить : $!";
```

Если используется формат с двумя аргументами для открытия канала в или из специальной команды "-",[1] то сначала неявно выполняется fork. (В систе-

[1] Можно представлять себе это как пропуск команды в вышеприведенных форматах с тремя аргументами.

мах, где отсутствует fork, это создает исключительную ситуацию. В системах Microsoft fork не поддерживалась до Perl 5.6.) В данном случае минус представляет новый дочерний процесс, являющийся копией родительского. Возвращаемое значение от этого ветвящегося open представляет собой ID дочернего процесса при рассмотрении его из родительского процесса, 0 при рассмотрении из дочернего процесса и неопределенное значение undef при неуспехе fork (в этом случае нет дочернего процесса). Например:

```
defined($pid = open(FROM_CHILD, "-|"))
        or die "невозможно выполнить : $!";

if ($pid) {
        @parent_lines = <FROM_CHILD>;  # код родителя
}
else {
        print STDOUT @child_lines;     # код потомка
}
```

Указатель файла ведет себя типичным для родительского процесса образом, но для дочернего процесса вход (или выход) родительского процесса передается по каналу из (или в) STDOUT (или STDIN) дочернего процесса. Дочерний процесс не видит открытого указателя файла родительского процесса. (На это указывает PID, равный 0.) Обычно следует использовать эту конструкцию вместо обычного open для канала, если требуется больший контроль над тем, как выполняется команда канала (например, при выполнении setuid), и нежелательно искать в командах оболочки метасимволы. Следующие открытия каналов примерно эквивалентны:

```
open FH,            "| tr   'a-z'  'A-Z'";        # канал в команду оболочки
open FH, "|-",        'tr', 'a-z', 'A-Z';         # канал в голую команду
open FH, "|-" or exec 'tr', 'a-z', 'A-Z' or die;  # канал в дочерний процесс
```

как и эти:

```
open FH,               "cat    -n   'file' |";     # канал из команды оболочки
open FH, "-|",         'cat', '-n', 'file';        # канал из голой команды
open FH, "-|" or exec 'cat', '-n', 'file' or die; # канал из дочернего процесса
```

Более искусное применение открытия с порождением потомков см. в разделах «Разговор с самим собой» главы 16 и «Зачистка окружения» главы 23 «Защита данных».

При запуске команды с помощью open необходимо выбрать ввод или вывод: "cmd|" для чтения или "|cmd" для записи. Нельзя использовать open для запуска команды, открывающей канал одновременно на вход и на выход, как, казалось бы, позволило запрещенное (в данное время) обозначение "|cmd|". Однако стандартные библиотечные программы IPC::Open2 и IPC::Open3 дают близкий эквивалент. Подробности о каналах с двумя концами см. в разделе «Межпроцессное взаимодействие» главы 16.

Можно также, в традиции оболочки Борна, указать *EXPR*, начинающееся с >&; в таком случае оставшаяся часть строки интерпретируется как имя указа-

теля файла (или дескриптор файла, если это число), который должен быть дублирован посредством системного вызова *dup2*(2).[1] Можно использовать & после >, >>, <, +>, +>> и +<. (Задаваемый режим должен соответствовать режиму исходного указателя файла.)

Это может понадобиться в том случае, если существует открытый указатель файла и есть желание создать еще один указатель, являющийся точной копией первого.

```
open(SAVEOUT, ">&SAVEERR") or die "невозможно дублировать SAVEERR: $!";
open(MHCONTEXT, "<&4")     or die "невозможно дублировать fd4: $!";
```

Это значит, что если функция рассчитывает получить имя файла, но вы не хотите давать ей имя файла, потому что уже открыли файл, то можете просто передать ей указатель файла с амперсандом впереди. Однако лучше всего использовать полностью квалифицированный указатель файла, поскольку функция может находиться в другом пакете:

```
somefunction("&main::LOGFILE");
```

Другой причиной, по которой может потребоваться дублировать указатель файла, является желание временно переадресовать текущий указатель, не потеряв при этом первоначального адресата. Вот сценарий, который запоминает, переадресовывает и восстанавливает STDOUT и STDERR:

```
#!/usr/bin/perl
open SAVEOUT, ">&STDOUT";
open SAVEERR, ">&STDERR";

open STDOUT, ">foo.out" or die "невозможно перенаправить stdout";
open STDERR, ">&STDOUT" or die "невозможно дублировать stdout";

select STDERR; $| = 1;         # включить автоматическую очистку буфера
select STDOUT; $| = 1;         # включить автоматическую очистку буфера

print STDOUT "stdout 1\n";     # эти потоки ввода-вывода распространяются
print STDERR "stderr 1\n";     # также в подпроцессы

system("some command");        # использует новые stdout/stderr

close STDOUT;
close STDERR;

open STDOUT, ">&SAVEOUT";
open STDERR, ">&SAVEERR";

print STDOUT "stdout 2\n";
print STDERR "stderr 2\n";
```

[1] Это (пока) не работает с объектами ввода-вывода на ссылках typeglob путем самооживления дескриптора файла, но всегда можно использовать fileno для получения дескриптора файла и его дублирования.

Если указателю файла или номеру его дескриптора предшествует комбинация &=, а не просто &, то вместо создания совершенно нового дескриптора файла Perl делает *FILEHANDLE* псевдонимом существующего дескриптора с помощью вызова библиотеки C *fdopen*(3). При этом несколько экономятся системные ресурсы, хотя в теперешние времена это не столь важно.

```
$fd = $ENV{"MHCONTEXTFD"};
open(MHCONTEXT, "<&=$fdnum")
        or die "couldn't fdopen descriptor $fdnum: $!";
```

Указатели файлов STDIN, STDOUT и STDERR всегда остаются открытыми при выполнении exec. Другие указатели файлов, по умолчанию, – нет. В системах, поддерживающих функцию fcntl, можно модифицировать флаг указателя файла для закрытия при выполнении exec.

```
use Fcntl qw(F_GETFD F_SETFD);
$flags = fcntl(FH, F_SETFD, 0)
        or die "Невозможно сбросить флаг закрытия при exeec для FH: $!\n";
```

См. также описание специальной переменной $^F ($SYSTEM_FD_MAX) в главе 28.

В одно- и двухаргументной формах open нужно проявлять осторожность при использовании строковой переменной в качестве имени файла, поскольку эта переменная может содержать произвольные таинственные символы (особенно если имя файла передано произвольными таинственными личностями через Интернет). Если не проявить осторожность, часть имени файла может быть интерпретирована как строка *MODE*, пробельный символ, который можно опустить, указание дублирования или минус. Вот один исторически интересный способ предосторожности:

```
$path =~ s#^(\s)#./$1#;
open (FH, "< $path\0") or die "невозможно открыть $path: $!";
```

Но и он взламывается несколькими способами. Лучше обратитесь к формату open с тремя аргументами, чтобы открывать произвольное имя файла аккуратно и без (дополнительной) угрозы для безопасности:

```
open(FH, "<", $path) or die "невозможно открыть $path: $!";
```

С другой стороны, если вы ищете действительно системный вызов *open*(2) в стиле C со всеми сопутствующими деталями и украшениями, возьмите sysopen:

```
use Fcntl;
sysopen(FH, $path, O_RDONLY) or die "невозможно открыть $path: $!";
```

Если система различает текстовые и двоичные файлы, то может потребоваться установить для указателя файла двоичный режим (или воздержаться от этого), чтобы избежать искажения файлов. В таких системах при задании текстового режима с двоичным файлом и двоичного режима с текстовым файлом результаты могут оказаться разочаровывающими.

Системы, в которых функция binmode нужна, отличаются от остальных по формату, используемому для текстовых файлов. В тех системах, которые ее не требуют, каждая строка завершается одним символом, соответствующим тому, что C понимает как перевод строки, – \n. В эту категорию попадают Unix и Mac OS. VMS, MVS, MS-всякие и S&M операционные системы прочих разновидностей рассматривают ввод-вывод текстовых и двоичных файлов по-разному, поэтому для них нужна binmode.

Или ее эквивалент. В версии Perl 5.6 можно задавать двоичный режим в функции open без отдельного вызова binmode. В составе аргумента *MODE* (но только в формате с тремя аргументами) можно задавать различные дисциплины ввода и вывода. Чтобы выполнить эквивалент binmode, используйте формат open с тремя аргументами и вставьте дисциплину :raw после остальных символов *MODE*:

```
open(FH, "<:raw", $path) or die "невозможно открыть $path: $!";
```

Поскольку эта функция очень новая, несомненно, к тому времени, когда эту книгу откроет читатель, будет иметься больше дисциплин, чем было, когда мы ее писали. Однако мы можем уверенно предсказать, что, по всей вероятности, будут иметься дисциплины, сходные со всеми или некоторыми, приведенными в табл. 29.2.

Таблица 29.2. Дисциплины ввода-вывода

Дисциплина	Значение
:raw	Двоичный режим; без обработки
:text	Обработка текста по умолчанию
:def	По умолчанию, объявляемая "use open"
:latin1	Файл должен быть в ISO-8859-1
:ctype	Файл должен быть в LC_CTYPE
:utf8	Файл должен быть в UTF-8
:utf16	Файл должен быть в UTF-16
:utf32	Файл должен быть в UTF-32
:uni	Угадать Unicode (UTF-*)
:any	Угадать Unicode/Latin1/LC_CTYPE
:xml	Использовать кодировку, указанную в файле
:crlf	Угадать перевод строки
:para	Режим абзацев
:slurp	Режим «заглатывания»

Будет возможность последовательно применять дисциплины (организовать стек дисциплин), когда это имеет смысл, поэтому, например, можно сказать:

```
open(FH, "<:para:crlf:uni", $path) or die "невозможно открыть $path: $!";
while ($para = <FH>) { ... }
```

При этом посредством дисциплин можно будет:

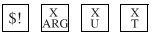
- считать какой-то формат Unicode и оттранслировать во внутренний формат Perl UTF-8;
- найти варианты последовательностей, завершающих строку, и оттранслировать их в `\n`;
- обработать файл, создав блоки размерами в параграф, подобно тому, как это делает `$/ = ""`.

Для того чтобы установить режим открытия по умолчанию (`:def`) в нечто отличное от `:text`, можно объявить это в начале файла с помощью директивы `open`:

```
use open IN => ":any", OUT => ":utf8";
```

На самом деле будет прекрасно, если когда-нибудь установится дисциплина по умолчанию `:text`. Она отлично передает дух девиза: «Будь терпим к тому, что принимаешь, и строг к тому, что создаешь».

opendir



```
opendir DIRHANDLE, EXPR
```

Функция открывает каталог с именем *EXPR* для работы с ним функций `readdir`, `telldir`, `seekdir`, `rewinddir` и `closedir`. Возвращает значение «истина» в случае успеха. Указателям каталогов отводится свое пространство имен, отличное от указателей файлов.

ord

```
ord EXPR
ord
```

Эта функция возвращает числовое значение (в ASCII, Latin-1 или Unicode) первого символа *EXPR*. Возвращаемое значение всегда беззнаковое. Если знак нужен, используйте `unpack('c', EXPR)`. Для того чтобы преобразовать все символы строки в список чисел, выполните `unpack('U*', EXPR)`.

our

```
our TYPE EXPR : ATTRIBUTES
our EXPR : ATTRIBUTES
our TYPE EXPR
our EXPR
```

Оператор `our` объявляет одну или более переменных как глобальные в охватывающем блоке, `eval` или файле. Это значит, что объявление `our` подчиняется таким же правилам видимости, как объявление `my`, но не создает новую закрытую переменную; оно просто разрешает свободное обращение к су-

ществующей в пакете глобальной переменной. Если перечислено несколько переменных, список должен быть помещен в круглые скобки.

Основная цель объявления `our` – скрыть переменную от действия объявления `use strict "vars"`; поскольку переменная маскируется под переменную типа `my`, можно использовать объявленную таким образом глобальную переменную, не квалифицируя ее именем ее пакета. Однако, подобно переменной `my`, это действует только в пределах лексической области видимости объявления `our`. В этом отношении оно отличается от директивы `use vars`, воздействующей на весь пакет, а не ограничивающейся лексической областью видимости.

`our` похоже на `my` и в том, что разрешается объявлять переменные с *TYPE* и *ATTRIBUTES*. Вот синтаксис:

```
our Dog $spot :ears(short) :tail(long);
```

На момент написания не вполне ясно, во что это выльется. Атрибуты могли бы воздействовать на глобальную или локальную интерпретацию `$spot`. С одной стороны, было бы более всего похоже на переменные `my`, если бы атрибуты охватывали текущее локальное представление `$spot`, не трогая другие представления глобальной переменной в иных местах. С другой стороны, если в одном модуле объявлено, что `$spot` – это `Dog`, а в другом объявлено, что `$spot` – это `Cat`, то можно прийти к мяукающим собакам и лающим котам. Это составляет предмет проводимых в настоящее время исследований,– таков витиеватый способ признания в том, что мы пока не понимаем того, что обсуждаем. (За исключением того, что мы знаем, что делать с объявлением *TYPE*, если переменная ссылается на псевдохеш, – см. раздел «Управление данными экземпляров» в главе 12.)

Другое сходство между `our` и `my` заключается в области видимости. Объявление `our` объявляет глобальную переменную, которая будет видима во всей лексической области видимости, даже пересекая границы пакетов. Пакет, в котором расположена переменная, определяется в месте нахождения объявления, а не в точке использования. Это значит, что имеет место следующее поведение, рассматриваемое как характерное свойство:

```
package Foo;
our $bar;       # $bar представляет собой $Foo::bar до конца лексической области
$bar = 582;

package Bar;
print $bar;     # выводит 582, как если бы "our" было "my"
```

Однако важным является различие, состоящее в том, что `my` создает новую закрытую переменную, а `our` делает видимой существующую глобальную переменную, особенно в присвоениях. Если объединить присвоение на этапе исполнения с объявлением `our`, значение, присвоенное глобальной переменной, не исчезнет, когда `our` выйдет из области видимости. Для этого потребуется `local`:

```
($x, $y) = ("один", "два");
print "перед блоком, x равно $x, y равно $y\n";
{
    our $x = 10;
    local our $y = 20;
    print "в блоке, x равно $x, y равно $y\n";
}
print "после блока, x равно $x, y равно $y\n";
```

Вывод будет таким:

```
перед блоком, x равно один, y равно два
в блоке, x равно 10, y равно 20
после блока, x равно 10, y равно два
```

Многократные объявления `our` в одной и той же лексической области допускаются, если они находятся в разных пакетах. Если они находятся в одном пакете, Perl выдает предупреждения, когда его об этом просят.

```
use warnings;
package Foo;
our $bar;          # объявляет $Foo::bar до конца лексической области
$bar = 20;

package Bar;
our $bar = 30;     # объявляет $Bar::bar до конца лексической области
print $bar;        # выводит 30

our $bar;          # выдает предупреждения
```

См. также `local`, `my` и раздел «Объявления с областью видимости» главы 4.

pack

```
pack TEMPLATE, LIST
```

Эта функция принимает список *LIST* обычных значений Perl, преобразует их в строку байт согласно шаблону *TEMPLATE* и возвращает эту строку. Список аргументов при необходимости дополняется или обрезается, т. е. если задать меньше аргументов, чем требует *TEMPLATE*, `pack` принимает дополнительные нулевые аргументы, а если задать больше аргументов, чем требует *TEMPLATE*, то лишние аргументы игнорируются. Нераспознаваемые элементы формата в *TEMPLATE* создадут исключительную ситуацию.

Шаблон описывает структуру строки как последовательность полей. Каждое поле представлено одним символом, который описывает тип кодируемого в нем значения. Например, символ формата `N` задает беззнаковое четырехбайтовое целое число в «тупоконечном» (big-endian) порядке байтов.

Поля упаковываются в порядке, задаваемом шаблоном. Например, чтобы упаковать в строку беззнаковое однобайтовое целое и значение с плавающей точкой с одинарной точностью, следует сказать:

```
$string = pack("Cf", 244, 3.14);
```

Первый байт возвращаемой строки имеет значение 244. В остальных байтах 3,14 кодируется как число одинарной точности с плавающей точкой. Конкретная кодировка числа с плавающей точкой зависит от аппаратной части компьютера.

При упаковке нужно учитывать некоторые важные вещи:

- тип данных (например, целое, с плавающей точкой, строка);
- диапазон значений (например, помещаются ли ваши целые числа в один, два, четыре или, может быть, даже восемь байтов; упаковываете ли вы 8-разрядные символы или символы Unicode);
- являются ли целые числа знаковыми или беззнаковыми;
- какая кодировка используется (родная, «остроконечная» (little-endian) или «тупоконечная» (big-endian) упаковка битов и байтов).

В табл. 29.3 перечислены символы формата и их значения. (В форматах встречаются и другие символы, которые будут описаны позже.)

Таблица 29.3. Символы шаблона для pack/unpack

Символ	Значение
a	Строка байт, дополняемая нулями
A	Строка байт, дополняемая пробелами
b	Битовая строка с возрастанием порядка разрядов в каждом байте (как vec)
B	Битовая строка с убыванием порядка разрядов в каждом байте
c	signed char (8-разрядное целое)
C	unsigned char (8- разрядное целое); смотрите U для Unicode
d	Число с плавающей запятой двойной точности в родном формате
f	Число с плавающей запятой одинарной точности в родном формате
h	Шестнадцатеричная строка, младший полубайт впереди
H	Шестнадцатеричная строка, старший полубайт впереди
i	Целое со знаком в родном формате
I	Целое без знака в родном формате
l	Длинное целое со знаком, всегда 32 разряда
L	Длинное целое без знака, всегда 32 разряда
n	16-разрядное короткое целое в «сетевом» (big-endian) порядке
N	32-разрядное длинное целое в «сетевом» (big-endian) порядке
p	Указатель на строку, завершающуюся нулем
P	Указатель на строку фиксированной длины
q	64-разрядное целое (quad) со знаком
Q	64-разрядное целое (quad) без знака
s	Короткое целое (short) со знаком, всегда 16 разрядов
S	Короткое целое (short) без знака, всегда 16 разрядов

Символ	Значение
u	Строка в кодировке uuencode
U	Номер символа Unicode
v	16-разрядное целое в порядке «VAX» (little-endian)
V	32-разрядное целое в порядке «VAX» (little-endian)
w	BER сжатое целое
x	Нулевой байт (проскочить на байт вперед)
X	Вернуться на байт
Z	Строка байт, завершающаяся нулем (и дополненная нулями)
@	Заполнить нулями до абсолютной позиции

Можно свободно располагать в шаблонах пробельные символы и комментарии. Комментарии начинаются с обычного символа # и простираются до первого символа перевода строки (если он есть) в *TEMPLATE*.

За каждой буквой может следовать число, означающее счетчик (*count*), интерпретируемое как число повторений или некоторого рода длина, в зависимости от формата. Для всех форматов, кроме a, A, b, B, h, H, P и Z, *count* означает число повторений, чтобы pack «заглатывал» из *LIST* указанное число значений. Значение счетчика * подразумевает все оставшиеся элементы.

Форматы a, A и Z захватывают только одно значение, но упаковывают его как строку длины *count*, дописывая при необходимости нулями или пробелами. При распаковке A отбрасывает замыкающие пробелы и нули, Z – все после первого нуля, и a возвращает литеральные данные, оставляя их в покое. При упаковке a и Z эквивалентны.

Аналогично форматы b и B упаковывают строку длиной *count* бит. Каждый байт входного поля генерирует 1 бит результата, исходя из наименее значимого разряда каждого входного байта (т. е. ord($byte) % 2). Это значит, что байты 0 и 1 генерируют биты 0 и 1. Каждая восьмерка байт с начала входной строки преобразуется в один байт на выходе. Если длина входной строки не кратна 8, остаток упаковывается, дополняясь нулями. Аналогично при распаковке лишние биты игнорируются. Если входная строка длиннее, чем нужно, лишние байты игнорируются. Значение счетчика * соответствует использованию всех байт входного поля. При распаковке биты преобразуются в строку, состоящую из 0 и 1.

Форматы h и H упаковывают строку из *count* полубайт (4-битовые группы, часто представляемые как шестнадцатеричные цифры).

Формат p упаковывает указатель на строку, заканчивающуюся нулем. Программист несет ответственность за то, чтобы эта строка не была временным значением (которое может быть удалено раньше, чем дело дойдет до применения упакованного результата). Формат P упаковывает указатель на структуру, размер которой задан в *count*. Если соответствующее значение для p или P представляет собой undef, создается нулевой указатель.

Символ / позволяет упаковывать и распаковывать строки, в которых упакованная структура содержит счетчик байт, за которым следует сама строка. Программист пишет: *length-item/string-item*. Элемент *length-item* может быть любой буквой шаблона для pack и описывает, как упаковано значение длины. Чаще всего используются упаковки целых чисел, такие как n (для строк Java), w (для ASN.1 или SNMP) и N (для Sun XDR). Элемент *string-item* должен быть (в момент написания книги) A*, a* или Z*. Для unpack длина строки извлекается из *length-item*, но если указать *, она игнорируется.

```
unpack 'C/a', "\04Gurusamy";        # дает 'Guru'
unpack 'a3/A* A*', '007 Bond  J ';  # дает (' Bond','J')
pack 'n/a* w/a*','hello,','world';  # дает "\000\006hello,\005world"
```

Элемент *length-item* не возвращается unpack явно. Добавление *count* к букве *length-item* вряд ли принесет какую-либо пользу, если только эта буква не A, a или Z. Упаковка с *length-item*, равным a или Z, может вводить нулевые символы (\0), которые Perl не считает законными в числовых строках.

За целочисленными форматами s, S, l и L может непосредственно следовать ! для обозначения родных коротких или длинных чисел вместо ровно 16 или 32 разрядов, соответственно. Сегодня это представляет проблему в основном на 64-разрядных платформах, где родные короткие и длинные числа могут иначе представляться своим компилятором C. (i! и I! тоже действуют, но только для полноты; они идентичны i и I.)

Фактический размер (в байтах) родных short (короткий), int (целый), long (длинный) и long long (удвоенный длинный) на платформе, где был скомпилирован Perl, тоже доступны посредством модуля Config:

```
use Config;
print $Config{shortsize},    "\n";
print $Config{intsize},      "\n";
print $Config{longsize},     "\n";
print $Config{longlongsize}, "\n";
```

Из того, что *Configure* известен размер long long, не обязательно следует, что вам доступны форматы q или Q. (В некоторых системах это так, но у вас, вероятно, не такая. Пока.)

Целочисленные форматы, превышающие в длину один байт (s, S, i, I, l и L) по природе своей непереносимы между процессорами, поскольку подчиняются родному порядку байт и очередности их сохранения. Для того чтобы получить переносимые упакованные целые, используйте форматы n, N, v и V; для них порядок записи байт и размер известны.

Числа с плавающей точкой имеют только родной формат машины. Из-за многообразия форматов с плавающей точкой и отсутствия стандартного «сетевого» представления нет средства для взаимообмена. Это значит, что упакованные числа с плавающей точкой, записанные на одной машине, могут не читаться на другой. Проблема остается, даже если обе машины используют арифметику с плавающей точкой IEEE, поскольку порядок записи байт в память («остро-» и «тупоконечность») не входит в спецификацию IEEE.

Perl внутренне использует представление с двойной точностью во всех вычислениях с плавающей точкой, поэтому при преобразовании из двойной точности в обычную и обратно теряется точность представления. Это значит, что `unpack("f", pack("f", $foo))` не всегда равно `$foo`.

Вы отвечаете за любые соображения, касающиеся выравнивания или дописывания, относящиеся к другим программам, особенно к тем программам, которые были созданы компилятором C со своими представлениями о том, как размещать структуру C для конкретной рассматриваемой архитектуры. Для этого при упаковке придется добавить достаточное количество `x`. Например, объявление C:

```
struct foo {
        unsigned char c;
        float f;
};
```

можно вывести в формате `"C x f"`, формате `"C x3 f"` или даже `"f C"`, – перечисляя лишь некоторые. Функции `pack` и `unpack` обрабатывают свои вход и выход как плоские последовательности байт, поскольку они не могут знать, куда идут или откуда поступают их байты.

Рассмотрим несколько примеров. Первая пара упаковывает числовые значения в байты:

```
$out = pack "CCCC", 65, 66, 67, 68;      # $out eq "ABCD"
$out = pack "C4", 65, 66, 67, 68;        # то же самое
```

Следующая команда делает то же самое с буквами Unicode, заключенными в кружок:

```
$foo = pack("U4",0x24b6,0x24b7,0x24b8,0x24b9);
```

Следующая команда делает аналогичную вещь, вводя пару нулей:

```
$out = pack "CCxxCC", 65, 66, 67, 68;    # $out eq "AB\0\0CD"
```

Упаковка коротких чисел не значит, что достигнута переносимость:

```
$out = pack "s2", 1, 2;    # "\1\0\2\0" на остроконечниках
                           # "\0\1\0\2" на тупоконечниках
```

При упаковке двоичных и шестнадцатеричных чисел *count* указывает количество бит или полубайт, а не создаваемых байт:

```
$out = pack "B32", "01010000011001010111001001101100";
$out = pack "H8", "5065726c";    # обе создают "Perl"
```

Длина поля `a` относится только к одной строке:

```
$out = pack "a4", "abcd", "x", "y", "z";      # "abcd"
```

Это ограничение можно обойти, указав несколько спецификаторов:

```
$out = pack "aaaa",  "abcd", "x", "y", "z";   # "axyz"
$out = pack "a" x 4, "abcd", "x", "y", "z";   # "axyz"
```

Формат a осуществляет заполнение нулями:

```
$out = pack "a14", "abcdefg";   # "abcdefg\0\0\0\0\0\0\0"
```

Этот шаблон упаковывает запись C struct tm (по крайней мере, в некоторых системах):

```
$out = pack "i9pl", gmtime(), $tz, $toff;
```

Обычно тот же формат можно использовать в функции unpack, хотя некоторые форматы действуют иначе, особенно a, A и Z.

Если вам нужно соединить вместе текстовые поля фиксированной ширины, применяйте pack с *TEMPLATE* из нескольких форматов A или a:

```
$string = pack("A10" x 10, @data);
```

Объединить текстовые поля переменной длины с разделителем позволяет функция join:

```
$string = join(" and ", @data);
$string = join("", @data);              # нулевой разделитель
```

Хотя во всех наших примерах в качестве шаблонов выступали литеральные строки, нет причин, по которым нельзя загружать шаблоны из дискового файла. На этой функции можно построить целую систему реляционной базы данных. (Мы не станем уточнять, как это будет характеризовать того, кто это сделает.)

package

```
package NAMESPACE
package
```

На самом деле это не функция, а объявление, которое говорит, что оставшаяся часть самой вложенной охватывающей области видимости принадлежит указанной таблице символов или пространству имен. (Область видимости объявления package, таким образом, такая же, как у объявлений my или our.) Внутри своей области видимости объявление заставляет компилятор разрешать все неквалифицированные глобальные идентификаторы путем поиска их в таблице символов объявленного пакета.

Объявление package действует только на глобальные переменные – в том числе те, с которыми использован оператор local, – не лексические переменные, созданные с помощью my. Оно воздействует только на неквалифицированные глобальные переменные, а глобальные переменные, которые квалифицированы именем своего собственного пакета, игнорируют текущий объявленный

пакет. Глобальные переменные, объявленные с помощью our, – неквалифицированные и потому учитывают текущий пакет, но только в точке объявления, после чего ведут себя как переменные my. То есть в оставшейся части своей лексической области видимости переменные our «прикрепляются» к пакету, используемому в точке объявления, даже если возникнет новое объявление пакета.

Обычно объявление package помещается в начале файла, который должен быть включен с помощью оператора require или use, но можно поместить его в любое место, где допускается команда. При создании обычного или объектно-ориентированного файла модуля пакету часто дают то же имя, что и файлу, чтобы избежать путаницы. (Принято также именовать такие пакеты с заглавной буквы, поскольку модули, названия которых начинаются с буквы на нижнем регистре, принято интерпретировать как модули директив.)

Можно переключаться в данный пакет более чем в одном месте. Это влияет только на выбор таблицы символов, используемой компилятором в оставшейся части блока. (Если компилятор видит другое объявление package на том же уровне, то новое объявление главенствует над прежним.) Предполагается, что ваша основная программа начинается с невидимого объявления package main.

Вы можете ссылаться на переменные, подпрограммы, указатели и форматы в других пакетах, квалифицируя идентификатор именем пакета и удвоенным двоеточием: $Package::Variable. Если имя пакета пустое, предполагается пакет main. Это значит, что $::sail эквивалентно $main::sail, а также $main'sail, что иногда еще можно встретить в старом коде.

Вот пример:

```
package main;       $sail = "hale and hearty";
package Mizzen;     $sail = "tattered";
package Whatever;
print "My main sail is $main::sail.\n";
print "My mizzen sail is $Mizzen::sail.\n";
```

Он выводит:

```
My main sail is hale and hearty.
My mizzen sail is tattered.
```

Таблица символов для пакета хранится в хеше, имя которого заканчивается удвоенным двоеточием. К примеру, таблица символов пакета main называется %main::. Поэтому к символу *main::sail можно также обратиться как к $main::{"sail"}.

Если пространство имен *NAMESPACE* опущено, то текущего пакета нет и все идентификаторы должны быть полностью квалифицированы или объявлены как лексические переменные. Это более строгое условие, чем use strict, поскольку распространяется и на имена функций.

Дополнительные сведения о пакетах см. в главе 10 «Пакеты». См. выше в этой главе описание my по другим вопросам областей видимости.

pipe

```
pipe READHANDLE, WRITEHANDLE
```

Как и соответствующий системный вызов, данная функция открывает пару связанных каналов – см. *pipe*(2). Этот вызов обычно осуществляется прямо перед `fork`, после которого программа чтения из канала должна закрыть *WRITEHANDLE*, а программа записи закрыть *READHANDLE*. (Иначе канал скроет EOF от программы чтения, когда будет закрыт программой записи.) Если устроить цикл канализированных процессов, может произойти взаимоблокировка, если не проявить особую осторожность. Кроме того, обратите внимание на то, что каналы Perl используют буферизацию стандартного ввода-вывода, поэтому может потребоваться установить `$|` (`$OUTPUT_AUTOFLUSH`) для *WRITEHANDLE*, чтобы очищать буфер после каждой операции вывода в зависимости от приложения – см. `select` (указатель выходного файла).

(Как и в `open`, если один из указателей файла не определен, он будет самооживлен.)

Вот небольшой пример:

```
pipe(README, WRITEME);
unless ($pid = fork) {  #               потомок
    defined $pid or die "невозможно создать потомок: $!";
    close(README);
    for $i (1..5) { print WRITEME "line $i\n" }
    exit;
}
$SIG{CHLD} = sub { waitpid($pid, 0) };
close(WRITEME);
@strings = <README>;
close(README);
print "Got:\n", @strings;
```

Обратите внимание на то, как записывающий код закрывает читающий конец, а код чтения закрывает записывающий конец. Нельзя организовать двустороннюю связь при помощи одного канала. Для этого необходимы два разных канала или системный вызов `socketpair`. См. раздел «Каналы» главы 16.

pop

```
pop ARRAY
pop
```

Эта функция обращается с массивом как со стеком – она выталкивает (удаляет) и возвращает последнее значение в массиве, укорачивая массив на один элемент. Если *ARRAY* опущен, функция выталкивает `@_` в лексической области видимости подпрограмм и форматов; она выталкивает `@ARGV` в области видимости файлов (обычно основной программы) или внутри лексических областей видимости, установленных конструкциями `eval` *STRING*, `BEGIN {}`, `CHECK {}`, `INIT {}` и `END {}`. Она оказывает такой же эффект, как:

```
$tmp = $ARRAY[$#ARRAY- -];
```

или:

```
$tmp = splice @ARRAY, -1;
```

Если в массиве нет элементов, pop возвращает undef. (Но не полагайтесь на это для того, чтобы определить, когда массив станет пуст, если он содержит значения undef!) См. также push и shift. Если требуется вытолкнуть более одного элемента, используйте splice.

Функция pop требует, чтобы ее первый аргумент был массивом, а не списком. Если просто нужен последний элемент списка, используйте

```
(LIST )[-1]
```

pos

```
pos SCALAR
pos
```

Эта функция возвращает место в *SCALAR*, где закончился последний поиск m//g в этом скаляре. Она возвращает смещение символа *после* последнего найденного. (Это равносильно length($`) + length($&).) Это смещение, по которому начнется следующий поиск m//g в этой строке. Запомните, что смещение начала строки равно 0. Например:

```
$graffito = "fee fie foe foo";
while ($graffito =~ m/e/g) {
    print pos $graffito, "\n";
}
```

выводит 2, 3, 7 и 11, т. е. смещения всех символов, следующих за "e". Функции pos можно присвоить значение, чтобы сообщить, где нужно начать следующий m//g:

```
$graffito = "fee fie foe foo";
pos $graffito = 4;  # Пропустить fee, начать с fie
while ($graffito =~ m/e/g) {
        print pos $graffito, "\n";
}
```

Здесь выводятся только 7 и 11. Утверждение регулярного выражения \G соответствует только положению, которое в настоящее время указывается pos для строки, в которой производится поиск. См. раздел «Позиции» главы 5.

print

```
print FILEHANDLE LIST
print LIST
print
```

Эта функция выводит строку или список строк, разделенных запятыми. Если установлена переменная $\ ($OUTPUT_RECORD_SEPARATOR), то ее содержимое выводится в конце списка. Функция возвращает значение «истина» в случае успеха и «ложь» в противном случае. *FILEHANDLE* может быть именем скалярной переменной (без индекса), и тогда переменная содержит либо имя фактического указателя файла, либо ссылку на какой-либо объект указателя файла. Как и любой другой косвенный объект, *FILEHANDLE* может также быть блоком, который возвращает такое значение:

```
print { $OK ? "STDOUT" : "STDERR" } "stuff\n";
print { $iohandle[$i] } "stuff\n";
```

Если *FILEHANDLE* представляет собой переменную, а очередная лексема – терм, он может быть неправильно принят за оператор, если не вставить знак + или не заключить аргументы в скобки, например:

```
print $a - 2;   # выводит $a - 2 в указатель файла по умолчанию (обычно STDOUT)
print $a (- 2); # выводит -2 в указатель файла, заданный в $a
print $a -2;    # тоже выводит -2 (таинственные правила синтаксического анализа:-)
```

Если *FILEHANDLE* опущен, то функция осуществляет вывод в указатель файла, выбранный в настоящий момент, первоначально STDOUT. Чтобы установить указатель выходного файла отличным от STDOUT, используйте операцию select *FILEHANDLE*.[1] Если *LIST* тоже опущен, функция выводит $_. Поскольку print принимает *LIST*, содержимое *LIST* вычисляется в списковом контексте. Поэтому, если сказать:

```
print OUT <STDIN>;
```

то будет выводиться не очередная строка из стандартного ввода, а все оставшиеся строки из стандартного ввода вплоть до конца файла, потому что именно это <STDIN> возвращает в списковом контексте. Если требуется другое, скажите:

```
print OUT scalar <STDIN>;
```

Кроме того, помня правило: «Все, что похоже на функцию, функцией и является», следите за тем, чтобы за ключевым словом print не следовала левая круглая скобка, если только вы не хотите, чтобы соответствующая правая круглая скобка завершала аргументы print, – вставьте знак + или заключите в круглые скобки все аргументы:

```
print (1+2)*3, "\n";            # НЕВЕРНО
print +(1+2)*3, "\n";           # ok
print ((1+2)*3, "\n");          # ok
```

[1] Поэтому в действительности STDOUT не является указателем файла по умолчанию для print. Это лишь значение по умолчанию значения по умолчанию для указателя файла.

printf

```
printf FILEHANDLE FORMAT, LIST
printf FORMAT, LIST
```

Эта функция выводит форматированную строку в *FILEHANDLE* или, если он опущен, выбранный в данное время текущий указатель файла, первоначально STDOUT. Первым элементом *LIST* должна быть строка, в которой указано, как форматировать остальные элементы. Функция аналогична функциям *printf* (3) и *fprintf* (3) из библиотеки C. Эта функция эквивалентна следующему:

```
print FILEHANDLE sprintf FORMAT, LIST
```

за исключением того, что $\ ($OUTPUT_RECORD_SEPARATOR) не добавляется. Если действует use locale, то символ, используемый в качестве десятичной точки в форматированных числах с плавающей точкой, определяется национальной установкой LC_NUMERIC.

Исключительная ситуация создается, только если в качестве аргумента *FILEHANDLE* задан недопустимый тип ссылки. Нераспознанные форматы передаются в целости. В обеих ситуациях выдаются предупредительные сообщения, если их вывод включен.

См. описания функций print и sprintf в данной главе. Описание sprintf содержит список спецификаций форматов. Нам следовало бы продублировать их здесь, но эта книга и без того представляет собой экологическую катастрофу.

Если опустить как *FORMAT*, так и *LIST*, то используется $_ – но в таком случае следовало остановить выбор на print. Не попадайтесь в ловушку, применяя printf там, где достаточно обычной print. Функция print более эффективна и менее подвержена ошибкам.

prototype

```
prototype FUNCTION
```

Возвращает прототип функции в виде строки (или undef, если у функции нет прототипа). *FUNCTION* представляет собой ссылку на функцию или имя функции, прототип которой требуется извлечь.

Если *FUNCTION* является строкой, начинающейся с CORE::, то оставшаяся часть принимается за имя встроенной функции Perl, и если такой встроенной функции нет, то создается исключительная ситуация. Если встроенная функция не *переопределяема* (например, qw//) или ее аргументы нельзя выразить прототипом (например, system), функция возвращает undef, поскольку встроенная функция ведет себя в действительности не так, как функция Perl. В противном случае возвращается строка, описывающая эквивалентный прототип.

push

```
push ARRAY, LIST
```

Эта функция рассматривает *ARRAY* как стек и выталкивает значения из списка *LIST* в конец *ARRAY*. Длина *ARRAY* возрастает на длину *LIST*. Функция возвращает эту новую длину. Функция `push` оказывает такое же воздействие, как:

```
foreach $value (listfunc()) {
    $array[++$#array] = $value;
}
```

или:

```
splice @array, @array, 0, listfunc();
```

но более эффективна (для вас и вашего компьютера). Можно использовать `push` в комбинации с `shift`, чтобы создать весьма эффективный по времени регистр сдвига или очередь:

```
for (;;) {
    push @array, shift @array;
    ...
}
```

См. также `pop` и `unshift`.

q/STRING/

```
q/STRING/
qq/STRING/
qr/STRING/
qw/STRING/
qx/STRING/
```

Обобщенные кавычки. См. раздел «Выберите собственные кавычки» главы 2. Сведения о статусе `qx//` см. в `readpipe`, а сведения о статусе `qr//` — в `m//`. См. также раздел «Управление процессом» главы 5.

quotemeta

```
quotemeta EXPR
quotemeta
```

Эта функция возвращает значение *EXPR*, помещая перед всеми символами, не являющимися буквой или цифрой, обратную косую черту. (То есть перед всеми символами, не совпадающими с `/[A-Za-z_0-9]/`, в возвращаемой строке ставится обратная косая черта независимо от национальных установок.) Это внутренняя функция, реализующая escape-последовательность `\Q` в интерполируемых контекстах (строках в двойных кавычках, обратных штрихах и шаблонах).

rand

```
rand EXPR
rand
```

Эта функция возвращает псевдослучайное число с плавающей точкой, большее или равное 0 и меньшее значения *EXPR*. (*EXPR* должно быть положительным.) Если выражение *EXPR* опущено, функция возвращает число с плавающей точкой между 0 и 1 (включая 0, но исключая 1). `rand` автоматически вызывает `srand`, если та еще не вызывалась. См. `srand`.

Чтобы получить случайное целое число, например, для моделирования игры в кости, подключите `int`, как в:

```
$roll = int(rand 6) + 1;        # $roll теперь число между 1 и 6
```

Поскольку Perl использует функцию псевдослучайных чисел из вашей собственной библиотеки C, например *random*(3) или *drand48*(3), качественное распределение не гарантируется. Если требуется более сильная случайность, например, для целей криптографии, можете обратиться к документации по *random* (4) (если у вас в системе есть устройство */dev/random* или */dev/urandom*), модулю `Math::TrulyRandom` в CPAN или прочитать хороший учебник по генерации псевдослучайных чисел на компьютерах, например второй том Кнута.[1]

read

```
read FILEHANDLE, SCALAR, LENGTH, OFFSET
read FILEHANDLE, SCALAR, LENGTH
```

Эта функция пытается прочесть *LENGTH* байт данных в переменную *SCALAR* из заданного указателя файла *FILEHANDLE*. Функция возвращает число прочитанных байт или 0, если достигнут конец файла. В случае ошибки она возвращает `undef`. *SCALAR* растет или сокращается в зависимости от фактически прочитанной длины. Смещение *OFFSET*, если оно задано, определяет, в какое место в переменной нужно начать помещать байты, поэтому можно осуществлять чтение в середину строки.

Чтобы скопировать данные из указателя файла `FROM` в указатель файла `TO`, можно сказать:

```
while (read(FROM, $buf, 16384)) {
    print TO $buf;
}
```

Противоположность `read` составляет просто `print`, которая уже знает длину записываемой строки и может записать строку любой длины. Было бы ошибкой использовать `write`, которая работает исключительно с форматами.

---

[1] Д. Кнут «Искусство программирования. Получисленные алгоритмы», т. 2, 3-е изд., издательство «Williams», 2001.

Функция Perl `read` реализована через функцию стандартного ввода-вывода *fread*(3), поэтому фактический системный вызов *read*(2) может прочесть больше байт, чем задано аргументом *LENGTH*, чтобы заполнить входной буфер, а *fread*(3) может выполнить более одного системного вызова *read*(2), чтобы заполнить буфер. Чтобы получить больший контроль, задайте реальный системный вызов с помощью `sysread`. Вызовы `read` и `sysread` не должны перемежаться, если только вы не заинтересованы в сильнодействующем колдовстве (или головной боли). Какую бы функцию вы не использовали, учитывайте, что при чтении файла, содержащего Unicode или другую многобайтовую кодировку, граница буфера может попасть на середину символа.

readdir

$! T X ARG X U

```
readdir DIRHANDLE
```

Функция читает записи каталога (являющиеся простыми именами файлов) из указателя каталога, открытого `opendir`. В скалярном контексте возвращает очередную запись каталога, если она есть, в противном случае она возвращает `undef`. В списковом контексте возвращает все оставшиеся записи в каталоге, что будет пустым списком, если записей больше нет. Например:

```
opendir(THISDIR, ".") or die "serious dainbramage: $!";
@allfiles = readdir THISDIR;
closedir THISDIR;
print "@allfiles\n";
```

Этот код выводит все файлы из текущего каталога в одну строку. Если вы хотите, чтобы не было записей `".."` и `"..."`, заколдуйте одну из них (ту, которую вы считаете менее нечитаемой):

```
@allfiles = grep { $_ ne '.' and $_ ne '..' } readdir THISDIR;
@allfiles = grep { not /^[.][.]?\z/ } readdir THISDIR;
@allfiles = grep { not /^\.{1,2}\z/ } readdir THISDIR;
@allfiles = grep !/^\.\.?\z/, readdir THISDIR;
```

Чтобы избежать вывода всех файлов `.*` (как в программе *ls*):

```
@allfiles = grep !/^\./, readdir THISDIR;
```

Чтобы получить только текстовые файлы:

```
@textfiles = grep -T, readdir THISDIR;
```

Но будьте внимательны с этим последним, потому что к результату `readdir` нужно снова «приклеить» часть, указывающую каталог, если это не текущий каталог, например:

```
opendir(THATDIR, $path) or die "can't opendir $path: $!";
@dotfiles = grep { /^\./ && -f } map { "$path/$_" } readdir(THATDIR);
closedir THATDIR;
```

readline

$! T X ARG

```
readline FILEHANDLE
```

Это внутренняя функция, реализующая оператор <FILEHANDLE>, но с ней можно работать непосредственно. Функция читает очередную запись из *FILEHANDLE*, что может быть именем указателя файла или выражением косвенного указателя файла, которое возвращает либо имя фактического указателя файла, либо ссылку на нечто напоминающее объект указателя файла, например typeglob. (Версии Perl до 5.6 принимают только typeglob.) В скалярном контексте каждый вызов считывает и возвращает очередную запись, пока не будет достигнут конец файла, и тогда очередной вызов возвращает undef. В списковом контексте readline читает записи, пока не будет достигнут конец файла, и возвращает список записей. Под «записью» обычно понимается строка текста, но при изменении значения $/ ($INPUT_RECORD_SEPARATOR) на иное, нежели установленное по умолчанию, этот оператор начинает «нарезать» текст по-другому. Аналогично некоторые дисциплины ввода, например :para (режим абзацев), возвращают записи не строчками, а другими фрагментами. Установление дисциплины :slurp (или неопределенного значения $/) влечет считывание файлов целиком (как одного фрагмента).

При считывании файлов целиком в скалярном контексте и чтении пустого файла readline в первый раз возвращает "", а каждый следующий раз возвращает undef. При считывании файлов целиком из магического указателя файла ARGV каждый файл возвращает один фрагмент (и здесь нулевые файлы возвращаются как ""), а затем, когда все файлы кончатся, следует одно значение undef.

Оператор <FILEHANDLE> более подробно описывается в разделе «Операторы ввода» главы 2.

```
$line = <STDIN>;
$line = readline(STDIN);          # то же самое
$line = readline(*STDIN);         # то же самое
$line = readline(\*STDIN);        # то же самое

open my $fh, "<&=STDIN" or die;
bless $fh => 'AnyOldClass';
$line = readline($fh);            # то же самое
```

readlink

$_ $! T X U

```
readlink EXPR
readlink
```

Эта функция возвращает имя файла, на который указывает символическая ссылка. Выражение *EXPR* должно иметь значением имя файла, последняя часть которого представляет собой символическую ссылку. Если это не символическая ссылка или символические ссылки не реализованы в файловой системе, или происходит какая-то системная ошибка, возвращается undef, и необходимо проверить код ошибки, находящийся в $!.

Имейте в виду, что возвращаемая символическая ссылка может быть относительной для указанного вами адреса. К примеру, можно сказать:

```
readlink "/usr/local/src/express/yourself.h"
```

а `readlink` может возвратить:

```
../express.1.23/includes/yourself.h
```

что нельзя непосредственно использовать как имя файла, если только текущим каталогом не является `/usr/local/src/express`.

readpipe

$! $? T X T X U

```
readpipe scalar EXPR
readpipe LIST (предложение)
```

Это внутренняя функция, реализующая конструкцию кавычек `qx//` (также известную как оператор галочки или обратного штриха). Иногда это удобно, когда требуется задать выражение *EXPR* так, что это не удобно в формате кавычек. Учтите, что в будущем мы можем поменять этот интерфейс с целью поддержки аргумента в виде списка, чтобы сделать функцию более похожей на `exec`, поэтому не рассчитывайте, что она будет по-прежнему предоставлять скалярный контекст для *EXPR*. Укажите `scalar` сами или попробуйте формат *LIST*. Кто знает, может быть, когда вы это будете читать, он уже заработает.

recv

$! T X ARG X RO X U

```
recv SOCKET, SCALAR, LEN, FLAGS
```

Эта функция получает сообщение на сокет. Она пытается получить *LENGTH* байт в переменную *SCALAR* из заданного указателя файла *SOCKET*. Функция возвращает адрес отправителя или `undef`, если возникает ошибка. *SCALAR* растет или сокращается в зависимости от фактически прочитанной длины. Функция принимает такие же флаги, как *recv*(2). См. раздел «Сокеты» главы 16.

redo

$@

```
redo LABEL
redo
```

Оператор `redo` запускает выполнение блока цикла с начала, не вычисляя повторно условия. Блок `continue`, если он имеется, не выполняется. Если метка *LABEL* опущена, то оператор относится к самому внутреннему охватывающему циклу. Этот оператор обычно применяется в программах, которым надо «обмануть» себя в отношении только что введенных данных:

```
# Цикл, соединяющий строчки с символом продолжения косая черта.
while (<STDIN>) {
```

```
    if (s/\\\n$// && defined($nextline = <STDIN>)) {
        $_ .= $nextline;
        redo;
    }
    print;  # или что-нибудь еще...
}
```

Посредством redo нельзя организовывать выход из блока, который возвращает значение, например eval {}, sub {} или do {}, и его не следует применять для выхода из операции grep или map. Если включен вывод предупреждений, то Perl предупреждает о redo цикла, находящегося вне текущей лексической области видимости.

Собственно блок семантически идентичен циклу, выполняемому один раз. Поэтому redo внутри такого блока превращает его в циклическую конструкцию. См. раздел «Управление циклом» главы 4.

ref

```
ref EXPR
ref
```

Оператор ref возвращает значение «истина», если *EXPR* является ссылкой, и «ложь» в противном случае. Возвращаемое значение зависит от типа того, на что указывает ссылка. В число встроенных типов входят:

```
SCALAR
ARRAY
HASH
CODE
GLOB
REF
LVALUE
IO::Handle
```

Если предмет ссылки освящен в пакет, то возвращается имя этого пакета. Можно представлять себе ref как оператор «typeof».

```
if (ref($r) eq "HASH") {
    print "r является ссылкой на хеш.\n";
}
elsif (ref($r) eq "Hump") {    # Нехорошо - см. ниже.
    print "r является ссылкой на объект Hump.\n";
}
elsif (not ref $r) {
    print "r вообще не является ссылкой.\n";
}
```

В ООП считается плохим стилем проверять класс объекта на равенство какому-либо конкретному имени, поскольку производный класс имеет другое имя, но ему разрешен доступ к методам базового класса. Лучше использовать метод isa класса UNIVERSAL следующим способом:

```
if ($r->isa("Hump") }
    print "r является ссылкой на объект Hump или подкласс.\n";
}
```

Лучше всего совсем не выполнять проверку, т. к. механизм объектов вообще не пошлет вашему методу объект, если сначала не убедится, что это допустимо. Дополнительные подробности см. в главах 8 и 12. См. также функцию `reftype` для директивы `use attributes` в главе 31.

rename

$! X T

```
rename OLDNAME, NEWNAME
```

Эта функция изменяет имя файла. Она возвращает значение «истина» в случае успеха и «ложь» в противном случае. Она не действует (обычно) через границы файловых систем, хотя в системе Unix это иногда компенсируется командой *mv*. Если файл *NEWNAME* уже существует, он уничтожается. В системах, отличных от Unix, могут быть дополнительные ограничения.

О переименованиях между различными файловыми системами см. в стандартном модуле `File::Copy`.

require

$_ $! $@ X T

```
require VERSION
require EXPR
require
```

Функция утверждает зависимость какого-либо рода для своего аргумента.

Если аргумент является строкой, `require` загружает и выполняет код Perl, находящийся в отдельном файле, имя которого задано строкой. Это похоже на выполнение файла посредством `do`, за исключением того, что `require` проверяет, загружен ли уже библиотечный файл, и создает исключительную ситуацию при возникновении трудностей. (Поэтому ее можно использовать для выражения зависимостей между файлами, не беспокоясь о возможности повторной компиляции.) Как и родственные `do` и `use`, `require` умеет выполнять поиск в маршрутах для включаемых файлов, хранимых в массиве `@INC`, и обновлять `%INC` в случае успеха. См. главу 28.

Файл должен возвращать «истину» в качестве последнего значения, что указывает на успешное выполнение имеющегося кода инициализации, поэтому принято заканчивать такой файл символами `1;`, если только вы не уверены, что он в любом случае возвращает истинное значение.

Если аргументом `require` является номер версии вида 5.6.2, то `require` требует, чтобы выполняющаяся в данный момент версия Perl была не ниже указанной. (Perl также принимает числа с плавающей точкой, такие как 5.005_03, для совместимости с более старыми версиями Perl, но использование такого формата не поощряется в данное время, поскольку он не понятен

людям из других культур.) Поэтому сценарий, которому требуется Perl версии 5.6, может иметь первой строкой:

```
require 5.6.0;              # или require v5.6.0
```

и более ранние версии Perl прервут свою работу. Однако, как и все require, это осуществляется на этапе исполнения. Для проверки во время компиляции можно сказать use 5.6.0. См. также описание $PERL_VERSION в главе 28.

Если аргумент require представляет собой голое имя пакета (см. package), require автоматически предполагает наличие суффикса .pm, что облегчает загрузку стандартных модулей. Такое поведение похоже на use, за исключением того, что происходит это на этапе исполнения, а не на этапе компиляции, и не вызывается метод import. Например, чтобы загрузить *Socket.pm*, не вводя символы в текущий пакет, скажите так:

```
require Socket;             # вместо "use Socket;"
```

Однако такого же результата можно добиться следующим кодом, который дополнительно выводит предупреждение на этапе компиляции, если *Socket.pm* не удается найти:

```
use Socket ();
```

Применение require с голым именем также заменяет все :: в имени пакета на системный разделитель каталогов, обычно /. Иными словами, если попытаться:

```
require Foo::Bar;           # прекрасное голое имя
```

то функция require ищет файл *Foo/Bar.pm* в каталогах, заданных в массиве @INC. Но если попробовать так:

```
$class = 'Foo::Bar';
require $class;             # $class не голое имя
```

или так:

```
require "Foo::Bar";         # литерал в кавычках - не голое имя
```

то функция require будет искать файл *Foo::Bar* в каталогах, заданных в массиве @INC, и будет жаловаться, если не сможет его там найти. В этом случае можно сделать так:

```
eval "require $class";
```

См. также do *FILE*, команду use, директиву use lib и стандартный модуль FindBin.

reset

```
reset EXPR
reset
```

Эту функцию обычно употребляют (или злоупотребляют ею) в начале цикла или блоке continue в конце цикла с целью очистить глобальные переменные или сбросить поиск ??, чтобы он мог снова работать. Выражение интерпретируется как список отдельных символов (дефисы могут использоваться для задания диапазонов). Все скалярные переменные, массивы и хеши, начинающиеся с одной из заданных букв, устанавливаются в свое первоначальное состояние. Если выражение опущено, поиски одного соответствия (?*PATTERN*?) сбрасываются и могут выполняться снова. Функция сбрасывает переменные и поиски только для текущего пакета. Она всегда возвращает истинное значение.

Чтобы сбросить все переменные "X", можно сказать:

```
reset 'X';
```

А для того чтобы сбросить все переменные на нижнем регистре, можно сказать:

```
reset 'a-z';
```

Наконец, чтобы сбросить все поиски ??, можно сказать:

```
reset;
```

Сброс "A-Z" в пакете main делать не рекомендуется, поскольку вы очистите свои глобальные массивы и хеши ARGV, INC, ENV и SIG.

Лексические переменные (создаваемые my) этой функцией не затрагиваются. Применение reset несколько предосудительно, поскольку оно может легко очистить целое пространство имен, и потому что использование самого оператора ?? несколько предосудительно.

См. также функцию delete_package() из стандартного модуля Symbol в разделе «Защищенные разделы» главы 23.

return

```
return EXPR
return
```

Этот оператор заставляет текущую подпрограмму (или eval, или do *FILE*) немедленно осуществить возврат с заданным значением. Попытка использования return вне указанных трех мест влечет за собой исключительную ситуацию. Обратите также внимание на то, что eval не может выполнять return от имени подпрограммы, которая вызвала eval.

Выражение *EXPR* может вычисляться в списковом, скалярном или пустом контексте в зависимости от того, как будет использоваться возвращаемое значение (что может измениться). Это значит, что передаваемое выражение будет вычисляться в контексте вызова подпрограммы. Если подпрограмма вызывается в скалярном контексте, то *EXPR* тоже вычисляется в скалярном контексте. Если подпрограмма вызывается в списковом контексте, то *EXPR* тоже

вычисляется в списковом контексте и может возвращать список. `return` без аргументов возвращает скалярное значение `undef` в скалярном контексте, пустой список `()` в списковом контексте и (естественно) вообще ничего в пустом контексте. Контекст вызова подпрограммы можно определить, находясь внутри подпрограммы, с помощью функции (неправильно названной) `wantarray`.

reverse

```
reverse LIST
```

В списковом контексте эта функция возвращает списковое значение, состоящее из элементов *LIST* в обратном порядке. Функцию можно использовать для создания убывающих последовательностей:

```
for (reverse 1 .. 10) { ... }
```

Поскольку хеши уплощаются в списки при передаче как *LIST*, `reverse` можно также применять для инвертирования хеша, в предположении, что все значения являются уникальными:

```
%barfoo = reverse %foobar;
```

В скалярном контексте функция конкатенирует все элементы *LIST* и затем возвращает результирующую строку в обратном порядке символов.

Небольшой совет: реверсирование списка, отсортированного пользовательской функцией, часто можно выполнить проще, если сначала отсортировать этот список в обратном направлении.

rewinddir

```
rewinddir DIRHANDLE
```

Устанавливает текущую позицию для программы `readdir` в *DIRHANDLE* на начало каталога. Эта функция может иметься не на всех машинах, поддерживающих `readdir`, — `rewinddir` умирает, если не реализована. Она возвращает значение «истина» в случае успеха и «ложь» в противном случае.

rindex

```
rindex STR, SUBSTR, POSITION
rindex STR, SUBSTR
```

Эта функция работает так же, как `index`, за исключением того, что возвращает позицию *последнего* вхождения *SUBSTR* в *STR* (обратный `index`). Функция возвращает `$[-1`, если *SUBSTR* не найдена. Поскольку `$[` практически всегда в наше время равна `0`, функция практически всегда возвращает `-1`. Параметр *POSITION*, если задан, представляет собой самую правую позицию, которую можно возвратить. Пройти по строке в обратном направлении можно так:

```
$pos = length $string;
while (($pos = rindex $string, $lookfor, $pos) >= 0) {
    print "Найдена в $pos\n";
    $pos- -;
}
```

rmdir

$_ $! X T

```
rmdir FILENAME
rmdir
```

Эта функция удаляет каталог, заданный в *FILENAME*, если каталог пустой. В случае успеха функция возвращает значение «истина», в противном случае она возвращает значение «ложь». См. также модуль `File::Path`, если вам нужно сначала удалить содержимое каталога и по какой-то причине вы не хотите выйти в оболочку и вызвать `rm -r`. (Например, у вас может не быть оболочки или команды *rm*, поскольку у вас еще нет PPT.)

s///

T X RO X T

```
s///
```

Оператор подстановки. См. раздел «Операторы поиска по шаблону» главы 5.

scalar

```
scalar EXPR
```

Эту псевдофункцию можно использовать с *LIST*, чтобы принудить к вычислению *EXPR* в скалярном контексте, если вычисление в списковом контексте приводит к другому результату. Например:

```
my ($nextvar) = scalar <STDIN>;
```

не дает `<STDIN>` прочесть все строчки из стандартного ввода перед присвоением, поскольку присвоение списку (даже созданному `my`) создает списковый контекст. (В данном примере без `scalar` первая строчка из `<STDIN>` все же была бы присвоена `$nextvar`, но последующие строчки были бы прочтены и отброшены, поскольку список, которому осуществляется присвоение, может получить только одно скалярное значение.)

Конечно, более простым и коротким способом было бы просто отбросить круглые скобки, изменив тем самым списковый контекст на скалярный:

```
my $nextvar = <STDIN>;
```

Поскольку функция `print` является списковым оператором, необходимо сказать:

```
print "Length is ", scalar(@ARRAY), "\n";
```

чтобы вывести длину массива `@ARRAY`.

Не существует функции "list", соответствующей scalar, поскольку на практике никогда не требуется принудительно осуществлять вычисление в списковом контексте. Это связано с тем, что любая операция, которой нужен *LIST*, уже бесплатно предоставляет списковый контекст для своих списковых аргументов.

Поскольку scalar является унарным оператором, то если вы случайно используете в качестве *EXPR* список, заключенный в круглые скобки, он будет вести себя как скалярное выражение запятой, вычисляя все элементы, кроме последнего, в пустом контексте и возвращая последний элемент, вычисленный в скалярном контексте. Это редко оказывается тем, что требуется. Следующая одна команда:

```
print uc(scalar(&foo,$bar)),$baz;
```

(а)морально эквивалентна таким двум:

```
&foo;
print(uc($bar),$baz);
```

См. подробности об операторе запятой в главе 2. См. в разделе «Прототипы» главы 6 дополнительно об унарных операторах.

seek

```
seek FILEHANDLE, OFFSET, WHENCE
```

Эта функция устанавливает файловый указатель для *FILEHANDLE*, точно так же, как вызов *fseek*(3) для стандартного ввода-вывода. Левая позиция в файле имеет смещение 0, а не 1. Кроме того, смещения относятся к позициям байт, а не номеров строчек. В целом, поскольку строки имеют различную длину, невозможно обратиться к строке с конкретным номером, не просмотрев весь файл до этой точки, если только не известно, что все строки имеют определенную длину, или не построен индекс, транслирующий номера строчек в смещения байтов. (Те же ограничения касаются позиций символов в файлах с переменной длиной кодировки символов: операционная система не знает, что такое символы, она понимает только байты.)

FILEHANDLE может быть выражением, значение которого дает имя фактического указателя файла или является ссылкой на нечто, напоминающее объект указателя файла. Функция возвращает значение «истина» в случае успеха и «ложь» в противном случае. Для удобства функция может рассчитывать смещения от различных позиций в файле. Значение *WHENCE* задает позицию в файле, используемую *OFFSET* в качестве отправной точки: 0 – начало файла; 1 – текущая позиция в файле или 2 – конец файла. *OFFSET* может быть отрицательным при *WHENCE*, равном 1 или 2. Если требуется использовать для *WHENCE* символические имена, можете прибегнуть к SEEK_SET, SEEK_CUR и SEEK_END из модулей IO::Seekable или POSIX либо, в версии Perl 5.6, из модуля Fcntl.

Если требуется установить позицию в файле для sysread или syswrite, не применяйте seek; буферизация стандартного ввода-вывода делает результат ее

действия на позицию в файле непредсказуемым и непереносимым. Используйте вместо этого sysseek.

Из-за правил и строгостей ANSI C в некоторых системах нужно выполнять seek при каждом переключении между чтением и записью. Среди прочего, при этом может вызываться функция библиотеки стандартного ввода-вывода *clearerr* (3). Полезно задать *WHENCE* равным 1 (SEEK_CUR) и OFFSET 0, чтобы позиция в файле не переместилась:

```
seek(TEST,0,1);
```

Одно интересное применение этой функции дает возможность следовать за растущими файлами, например:

```
for (;;) {
    while (<LOG>) {
        grok($_);           # Обработать текущую строку.
    }

    sleep 15;
    seek LOG,0,1;           # Сбросить ошибку конца файла.
}
```

Последний вызов seek сбрасывает ошибку конца файла, не перемещая текущую позицию в файле. В зависимости от того, насколько стандартной окажется реализация ввода-вывода в конкретной библиотеке C, может понадобиться что-то еще, например:

```
for (;;) {
    for ($curpos = tell FILE; <FILE>; $curpos = tell FILE) {
        grok($_);           # Обработать текущую строку.
    }

    sleep $for_a_while;
    seek FILE, $curpos, 0; # Сбросить ошибку конца файла.
}
```

Аналогичная стратегия может применяться для запоминания адресов seek каждой строки в массиве.

seekdir

```
seekdir DIRHANDLE, POS
```

Эта функция устанавливает текущую позицию для следующего обращения к readdir в *DIRHANDLE*. *POS* должно быть значением, возвращенным telldir. Для этой функции действуют те же предостережения относительно возможного сжатия каталога, как и для соответствующей программы системной библиотеки. Функция может быть реализована не везде, где есть readdir. Она наверняка не реализована там, где нет readdir.

select (output filehandle)

X ARG

```
select FILEHANDLE
select
```

В силу исторических причин есть два оператора select, совершенно не связанных друг с другом. О втором рассказывается в следующем разделе. Эта версия оператора select возвращает выбранный в настоящее время указатель выходного файла и, если задан *FILEHANDLE*, устанавливает текущий указатель файла для выдачи. Это влечет два результата: во-первых, write или print без указателя файла по умолчанию осуществляются в этот *FILEHANDLE*. Во-вторых, специальные переменные, связанные с выдачей, будут ссылаться на этот указатель выходного файла. Например, если требуется установить одинаковый формат верхнего колонтитула для более чем одного указателя выходного файла, можно поступить так:

```
select REPORT1;
$^ = 'MyTop';

select REPORT2;
$^ = 'MyTop';
```

Но обратите внимание, что при этом REPORT2 остается текущим выбранным указателем файла. Такое поведение можно рассматривать как антисоциальное, поскольку оно может запутать команды print или write других программ. Правильно написанные библиотечные программы оставляют на выходе текущий указатель файла таким же, каким он был при вводе. Чтобы поддерживать такой режим, *FILEHANDLE* может быть выражением, значение которого дает имя фактического указателя файла. Поэтому можно сохранять и восстанавливать текущий выбранный указатель файла таким образом:

```
my $oldfh = select STDERR; $| = 1; select $oldfh;
```

или идиоматически, но несколько неясно:

```
select((select(STDERR), $| = 1)[0])
```

Этот пример действует, создавая список, состоящий из значений, возвращаемых select(STDERR) (которая в качестве побочного эффекта выбирает STDERR), и $| = 1 (что всегда 1), но устанавливает автоматическую очистку буфера для выбранного текущим STDERR в качестве побочного эффекта. Первый элемент этого списка (прежний выбранный указатель файла) используется теперь в качестве аргумента для внешнего оператора select. Необычно, правда? Это результат, который получается, если знать Lisp настолько, что это становится опасным.

Можно также использовать стандартный модуль SelectSaver, чтобы автоматически восстанавливать прежний select после выхода из области видимости.

Однако после того как мы все это изложили, нужно заметить, что в наше время редко требуется применять такую разновидность select, поскольку

для большинства специальных переменных, которые вам хотелось бы установить, есть методы в объектно-ориентированной оболочке, которые это сделают. Поэтому вместо непосредственного присвоения $| можно сказать:

```
use IO::Handle;             # К сожалению, это *не* маленький модуль.
STDOUT->autoflush(1);
```

А предыдущий пример форматирования можно записать так:

```
use IO::Handle;
REPORT1->format_top_name("MyTop");
REPORT2->format_top_name("MyTop");
```

select (готовые дескрипторы файлов)

```
select RBITS, WBITS, EBITS, TIMEOUT
```

Оператор select с четырьмя аргументами совершенно не связан с описанным выше оператором select. Этот оператор используется для того, чтобы определить, которые (если такие есть) из ваших дескрипторов файлов готовы для осуществления ввода или вывода, или чтобы сообщить о наличии исключительного условия. (Благодаря этому можно избежать необходимости проводить опрос.) Этот оператор обращается к системному вызову *select*(2) с заданными программистом битовыми масками, которые можно создать с помощью fileno и vec, например:

```
$rin = $win = $ein = "";

vec($rin, fileno(STDIN), 1)  = 1;
vec($win, fileno(STDOUT), 1) = 1;

$ein = $rin | $win;
```

Если необходимо выполнить select для нескольких указателей файлов, можно написать такую подпрограмму:

```
sub fhbits {
    my @fhlist = @_;
    my $bits;
    for (@fhlist) {
        vec($bits, fileno($_), 1) = 1;
    }
    return $bits;
}

$rin = fhbits(qw(STDIN TTY MYSOCK));
```

Для того чтобы использовать битовые маски многократно (а это более эффективно), обычно используется идиома:

```
($nfound, $timeleft) =
    select($rout=$rin, $wout=$win, $eout=$ein, $timeout);
```

Или осуществляется блокировка, пока не станет доступен какой-нибудь дескриптор файла:

```
$nfound = select($rout=$rin, $wout=$win, $eout=$ein, undef);
```

Как видите, вызов select в скалярном контексте просто возвращает $nfound — число найденных готовых дескрипторов.

Прием $wout=$win работает, поскольку значением присвоения является его левая часть; поэтому $wout разрушается сначала присвоением, а затем select, в то время как $win остается неизменной.

Любые из аргументов могут также представлять собой undef, и тогда они игнорируются. Значение *TIMEOUT*, если оно не undef, выражается в секундах и может быть дробным. (Тайм-аут 0 действует как опрос.) Не многие реализации способны возвращать $timeleft. И тогда они возвращают значение $timeleft, равное переданному $timeout.

Стандартный модуль IO::Select предоставляет более дружественный пользователю интерфейс к select, в основном благодаря тому, что берет на себя всю работу по созданию битовой маски.

Одно из применений select состоит в паузах с большей точностью времени, чем позволяет sleep. Следует указать undef для всех битовых масок. Поэтому для паузы в (по крайней мере) 4,75 секунды используйте:

```
select undef, undef, undef, 4.75;
```

(В некоторых системах, отличных от Unix, тройной undef может не работать, и может понадобиться подделать по крайней мере одну битовую маску действующего дескриптора, который никогда не будет готов.)

Вряд ли следует смешивать буферизованный ввод-вывод (как в случае read или <HANDLE>) с select, за исключением случаев, разрешенных POSIX, и даже в этом случае только на подлинных системах POSIX. Применяйте вместо этого sysread.

semctl

```
semctl ID, SEMNUM, CMD, ARG
```

Эта функция вызывает функцию System V IPC *semctl*(2). Возможно, придется сначала сказать use IPC::SysV, чтобы получить правильные определения констант. Если *CMD* имеет значение IPC_STAT или GETALL, то *ARG* должна быть переменной, которая содержит возвращаемую структуру semid_ds или массив значений семафоров. Как и для функций ioctl и fcntl, возвращаемыми значениями являются undef для ошибки, "0 but true" для нуля и фактическое возвращенное значение в противном случае.

См. также модуль IPC::Semaphore. Эта функция доступна только на машинах, поддерживающих System V IPC.

semget

```
semget KEY, NSEMS, FLAGS
```

Эта функция обращается к системному вызову System V IPC *semget*(2). Перед вызовом следует выполнить `use IPC::SysV`, чтобы получить правильные определения констант. Функция возвращает ID семафора или `undef` в случае ошибки.

См. также модуль `IPC::Semaphore`. Эта функция доступна только на машинах, поддерживающих System V IPC.

semop

```
semop KEY, OPSTRING
```

Эта функция обращается к системному вызову System V IPC *semop*(2), чтобы выполнить такие операции с семафорами, как передача сигналов и ожидание. Перед вызовом следует выполнить `use IPC::SysV`, чтобы получить правильные определения констант.

OPSTRING должна быть упакованным массивом структур `semop`. Каждую структуру `semop` можно создать, сказав `pack("s*", $semnum, $semop, $semflag)`. Число операций с семафорами определяется длиной *OPSTRING*. Функция возвращает значение «истина» в случае успеха и «ложь» при возникновении ошибки.

Следующий код ждет семафор `$semnum` с идентификатором (id) семафора `$semid`:

```
$semop = pack "s*", $semnum, -1, 0;
semop $semid, $semop or die "Semaphore trouble: $!\n";
```

Для передачи сигнала семафору просто замените `-1` на `1`.

См. раздел «System V IPC» главы 16. См. также модуль `IPC::Semaphore`. Эта функция доступна только на машинах, поддерживающих System V IPC.

send

```
send SOCKET, MSG, FLAGS, TO
send SOCKET, MSG, FLAGS
```

Эта функция посылает сообщение на сокет. Она принимает некоторые сигналы, как и системный вызов с тем же именем, – см. *send*(2). Для несоединенных сокетов нужно указать адрес отправки *TO*, что заставляет функцию Perl *send* работать как *sendto*(2). Системный вызов C *sendmsg*(2) в настоящее время не реализован в стандартном Perl. Функция `send` возвращает число отправленных байтов или `undef`, если возникла ошибка.

(Некоторые системы, отличные от Unix, неверно рассматривают сокеты как отличные от обычных дескрипторов файла, в результате чего приходится всегда использовать `send` и `recv` с сокетами вместо более удобных операторов стандартного ввода-вывода.)

Ошибка, которую по крайней мере один из нас часто совершает, состоит в том, что путаются `send` Perl и `send` C в записи:

```
send SOCK, $buffer, length $buffer      # НЕВЕРНО
```

Этот код непонятным образом приводит к ошибке в зависимости от связи между длиной строки и битами *FLAGS*, которые ждет система. Примеры см. в разделе «Передача сообщений» главы 16.

setpgrp

$! X T X U

```
setpgrp PID, PGRP
```

Функция устанавливает текущую группу процесса (*PGRP*) для заданного *PID* (используйте *PID*, равный 0, для текущего процесса). Вызов `setpgrp` создаст исключительную ситуацию, если будет выполнен на машине, где не реализован *setpgrp*(2). Будьте осторожны: некоторые системы игнорируют передаваемые аргументы и всегда выполняют `setpgrp(0, $$)`. К счастью, именно эти аргументы обычно и требуется передать. Если аргументы опущены, то по умолчанию они равны `0,0`. В версии BSD 4.2 функция `setpgrp` не принимала никаких аргументов, но в BSD 4.4 это синоним для функции `setpgid`. Для лучшей переносимости (согласно некоторому ее определению) используйте функцию `setpgid` в модуле `POSIX` непосредственно. Если в действительности кто-то пытается сделать демон из своего сценария, ему стоит подумать также о применении функции `POSIX::setsid()`. Обратите внимание, что POSIX-версия `setpgrp` не принимает аргументы, поэтому только `setpgrp(0,0)` является действительно переносимой.

setpriority

$! X T X U

```
setpriority WHICH, WHO, PRIORITY
```

Эта функция устанавливает текущий приоритет *PRIORITY* для процесса, группы процессов или пользователя в соответствии с *WHICH* и *WHO*. См. *setpriority*(2). Вызов `setpriority` создаст исключительную ситуацию, если будет выполнен на машине, на которой не реализован *setpriority*(2). Чтобы понизить приоритет процесса на четыре единицы (то же, что выполнение процесса с *nice*(1)), сделайте так:

```
setpriority 0, 0, getpriority(0, 0) + 4;
```

Интерпретация заданного приоритета может различаться в зависимости от операционной системы. Некоторые приоритеты могут быть недоступны для непривилегированных пользователей.

См. также модуль `BSD::Resource` в CPAN.

setsockopt

$! X ARG X U

```
setsockopt SOCKET, LEVEL, OPTNAME, OPTVAL
```

Эта функция устанавливает требуемые параметры сокета. Функция возвращает undef при ошибке. *LEVEL* указывает, на какой уровень протокола нацелен вызов, или SOL_SOCKET для самого сокета поверх всех уровней. *OPTVAL* можно задать как undef, если вы не хотите передавать аргумент. Часто для сокета устанавливается опция SO_REUSEADDR, чтобы обойти проблему, возникающую при невозможности привязаться к конкретному адресу, когда предыдущее соединение TCP на этом порту все еще принимает решение о закрытии. Это выглядит так:

```
use Socket;
socket(SOCK, ...) or die "Невозможно создать сокет: $!\n";
setsockopt(SOCK, SOL_SOCKET, SO_REUSEADDR, 1)
        or warn "Невозможно выполнить setsockopt: $!\n";
```

См. в *setsockopt*(2) другие возможные значения.

shift

```
shift ARRAY
shift
```

Эта функция сдвигает первое значение массива и возвращает его, укорачивая массив на единицу и смещая все вниз. (Или вверх, в зависимости от того, как вы зрительно представляете себе массив. Мы предпочитаем слева.) Если в массиве нет элементов, функция возвращает undef.

Если *ARRAY* опущен, то функция сдвигает @_ в лексической области видимости подпрограмм и форматов; она сдвигает @ARGV в областях видимости файлов (обычно основной программы) или в лексических областях видимости, установленных конструкциями eval *STRING*, BEGIN {}, CHECK {}, INIT {} и END {}.

Подпрограммы часто начинают работу с того, что копируют свои аргументы в лексические переменные, для чего может применяться shift:

```
sub marine {
        my $fathoms = shift;  # глубина
        my $fishies = shift;  # число рыб
        my $o2      = shift;  # содержание кислорода
        # ...
}
```

Также shift применяется для обработки аргументов в начале программы:

```
while (defined($_ = shift)) {
        /^[^-]/     && do { unshift @ARGV, $_; last };
        /^-w/       && do { $WARN = 1;         next };
        /^-r/       && do { $RECURSE = 1;      next };
        die "Неизвестный аргумент $_\n";
}
```

Для обработки аргументов программ можно также использовать модули Getopt::Std и Getopt::Long.

См. также функции unshift, push, pop и splice. Функции shift и unshift делают с левым концом массива то же самое, что pop и push делают с правым.

shmctl

```
shmctl ID, CMD, ARG
```

Эта функция вызывает системный вызов System V IPC *shmctl*(2). Перед вызовом следует выполнить use IPC::SysV, чтобы загрузить правильные определения констант.

Если *CMD* имеет значение IPC_STAT, то *ARG* должен быть переменной, в которую будет помещена возвращаемая структура shmid_ds. Подобно ioctl и fcntl, функция возвращает undef в случае ошибки, "0 but true" для нуля и фактически возвращенное значение в остальных случаях.

Эта функция доступна только на машинах, поддерживающих System V IPC.

shmget

```
shmget KEY, SIZE, FLAGS
```

Эта функция вызывает системный вызов System V IPC *shmget*(2). Возвращает ID сегмента совместно используемой памяти или undef при возникновении ошибки. Перед вызовом следует выполнить use SysV::IPC.

Эта функция доступна только на машинах, поддерживающих System V IPC.

shmread

```
shmread ID, VAR, POS, SIZE
```

Осуществляет чтение из сегмента совместно используемой памяти *ID* с позиции *POS* размером *SIZE* (путем прикрепления к нему, копирования из него данных и открепления от него). *VAR* указывает переменную, в которую будут помещены прочтенные данные. Функция возвращает значение «истина» в случае успеха и «ложь» при возникновении ошибки.

Эта функция доступна только на машинах, поддерживающих System V IPC.

shmwrite

```
shmwrite ID, STRING, POS, SIZE
```

Эта функция осуществляет запись в сегмент совместно используемой памяти *ID* с позиции *POS* размером *SIZE* (путем прикрепления к нему, копирования в него данных и открепления от него). Если *STRING* оказывается слишком длинной, используются только *SIZE* байт; если *STRING* слишком короткая, нули дописываются, чтобы заполнить *SIZE* байт. Функция возвращает значение «истина» в случае успеха и «ложь» при возникновении ошибки.

Эта функция доступна только на машинах, поддерживающих System V IPC. (Вы, наверное, уже устали это читать, – а мы устали это говорить.)

shutdown

```
shutdown SOCKET, HOW
```

Функция закрывает соединение с сокетом способом, описываемым *HOW*. Если *HOW* имеет значение 0, дальнейший прием запрещен. Если *HOW* имеет значение 1, дальнейшая отправка запрещена. Если *HOW* имеет значение 2, запрещено все.

```
shutdown(SOCK, 0);    # больше не читать
shutdown(SOCK, 1);    # больше не писать
shutdown(SOCK, 2);    # больше никакого ввода-вывода
```

Это удобно при работе с сокетами, когда необходимо сообщить другой стороне, что закончена запись, но не закончено чтение, или наоборот. Это также более настойчивая форма закрытия, поскольку она отключает также все копии дескрипторов файлов, имеющиеся в ответвившихся процессах.

Представьте себе сервер, которому нужно прочесть запрос клиента до получения конца файла, а затем отправить ответ. Если клиент вызывает `close`, то сокет становится недоступным для ввода-вывода, поэтому получить ответ невозможно. Вместо этого клиент должен использовать `shutdown` и «полузакрыть» соединение:

```
print SERVER "my request\n";        # отправить какие-то данные
shutdown(SERVER, 1);                # послать eof; больше не записывать
$answer = <SERVER>;                 # но можно продолжать чтение
```

(Если вы попали сюда в поисках того, как остановить систему, то вам придется для этого выполнить внешнюю программу. См. `system`.)

sin

```
sin EXPR
sin
```

Извините, ничего коварного в этом операторе нет. Он просто возвращает синус выражения *EXPR* (выраженного в радианах).

Для выполнения операции, обратной синусу, можно вызвать функцию `asin` из модуля `Math::Trig` или `POSIX` либо использовать следующее соотношение:

```
sub asin { atan2($_[0], sqrt(1 - $_[0] * $_[0])) }
```

sleep

```
sleep EXPR
sleep
```

Эта функция заставляет сценарий уснуть (сделать паузу) на *EXPR* секунд либо навсегда, если не задано *EXPR*, и возвращает число истекших секунд паузы. Паузу можно прервать, послав процессу SIGALRM. В некоторых более старых системах пауза может продолжаться на одну секунду меньше, чем вы запросили, в зависимости от способа подсчета секунд. Большинство современных систем спит столько, сколько положено. Однако может оказаться, что сон продолжается дольше чем нужно, поскольку в загруженной многозадачной системе планирование данного процесса может задержаться. Если в системе имеется вызов select (готовых дескрипторов файлов), то с его помощью можно получить лучшее разрешение времени. На некоторых системах Unix можно также выполнить syscall для вызова программ *getitimer* (2) и *setitimer* (2). Чередование вызовов alarm и sleep может оказаться недопустимым, поскольку sleep часто реализуется с помощью alarm.

См. также функцию sigpause в модуле POSIX.

socket

$! X ARG X T X U

```
socket SOCKET, DOMAIN, TYPE, PROTOCOL
```

Эта функция открывает сокет заданного вида и прикрепляет его к указателю файла *SOCKET*. *DOMAIN*, *TYPE* и *PROTOCOL* задаются так же, как в *socket*(2). Если *SOCKET* не определен, он самооживляется. До вызова этой функции в программе должна быть строка:

```
use Socket;
```

Она поставляет нужные константы. Функция возвращает значение «истина» в случае успешного выполнения. Примеры см. в разделе «Сокеты» главы 16.

В системах, которые поддерживают для файлов флаг закрытия при выполнении, этот флаг устанавливается для заново открытого дескриптора файла в соответствии со значением $^F. См. переменную $^F ($SYSTEM_FD_MAX) в главе 28.

socketpair

$! X ARG X T X U

```
socketpair SOCKET1, SOCKET2, DOMAIN, TYPE, PROTOCOL
```

Эта функция создает неименованную пару сокетов указанного типа в указанном домене. *DOMAIN*, *TYPE* и *PROTOCOL* задаются так же, как в *socketpair*(2). Если какой-либо аргумент сокета не определен, он будет самооживлен. Функция возвращает значение «истина» в случае успеха и «ложь» в противном случае. В системах, где *socketpair*(2) не реализован, вызов этой функции создает исключительную ситуацию.

Эта функция обычно вызывается непосредственно перед fork. Один из возникающих процессов должен закрыть *SOCKET1*, а второй должен закрыть *SOCKET2*. Эти сокеты можно использовать двунаправленным образом, в отличие от указателей файлов, создаваемых функцией pipe. На некоторых системах pipe определяется через socketpair, и тогда вызов pipe(Rdr, Wtr) является, в сущности, следующим:

```
use Socket;
socketpair(Rdr, Wtr, AF_UNIX, SOCK_STREAM, PF_UNSPEC);
shutdown(Rdr, 1);        # запись читающему процессу запрещена
shutdown(Wtr, 0);        # чтение пишущему процессу запрещено
```

В системах, которые поддерживают для файлов флаг закрытия при выполнении, этот флаг устанавливается для заново открытого дескриптора файла в соответствии со значением `$^F`. См. переменную `$^F` (`$SYSTEM_FD_MAX`) в главе 28. См. также пример в конце раздела «Двунаправленная связь» главы 16.

sort

```
sort USERSUB LIST
sort BLOCK LIST
sort LIST
```

Эта функция сортирует *LIST* и возвращает отсортированное списковое значение. По умолчанию она сортирует в стандартном порядке сравнения строк (неопределенные значения идут перед определенными нулевыми строками, за которыми идет все остальное). Если действует директива `use locale`, то `sort` *LIST* сортирует *LIST* в соответствии с текущими национальными установками сравнения.

USERSUB, если присутствует, представляет собой имя подпрограммы, которая возвращает целое число, меньшее, равное или большее 0, в зависимости от того, как должны быть упорядочены элементы списка. (Можно использовать для выполнения трехстороннего сравнения чисел и строк удобные операторы `<=>` и `cmp`.) Если *USERSUB* задана, но эта функция не определена, `sort` создает исключительную ситуацию.

С целью повышения производительности обычный код вызова подпрограмм минуется, что оказывает следующий эффект: подпрограмма не должна быть рекурсивной (нельзя также выходить из блока или программы с помощью операторов управления циклом), а два сравниваемые элемента не передаются в подпрограмму через `@_`, временно присваиваясь глобальным переменным `$a` и `$b` пакета, в который была компилирована `sort` (см. приведенные ниже примеры). Переменные `$a` и `$b` являются псевдонимами реальных величин, поэтому не модифицируйте их в подпрограмме.

Подпрограмма сравнения должна корректно вести себя. Если она возвращает несовместные результаты (сообщая, например, иногда, что `$x[1]` меньше `$x[2]`, а иногда утверждая обратное), то результаты не вполне определены. (Это еще одна причина, по которой не следует модифицировать `$a` и `$b`.)

Аргумент *USERSUB* может быть именем скалярной переменной (без индексов), и тогда он задает символическую или жесткую ссылку на подпрограмму, которая должна использоваться. (Символическое имя, а не жесткая ссылка допускается даже тогда, когда действует директива `use strict 'refs'`.) Вместо *USERSUB* можно задать блок в качестве анонимной подставляемой (inline) подпрограммы сортировки.

Чтобы выполнить обычную сортировку чисел, скажите:

```
sub numerically { $a <=> $b }
@sortedbynumber = sort numerically 53,29,11,32,7;
```

Чтобы выполнить сортировку в порядке убывания, можно просто применить после сортировки `reverse` либо поменять местами `$a` и `$b` в подпрограмме сортировки:

```
@descending = reverse sort numerically 53,29,11,32,7;
sub reverse_numerically { $b <=> $a }
@descending = sort reverse_numerically 53,29,11,32,7;
```

Чтобы выполнить сортировку строк без учета регистра, перед сравнением пропустите `$a` и `$b` через `lc`:

```
@unsorted = qw/sparrow Ostrich LARK catbird blueJAY/;
@sorted = sort { lc($a) cmp lc($b) } @unsorted;
```

(В случае работы с Unicode `lc` для канонизации регистра может быть предпочтительнее `uc`, поскольку в некоторых языках заглавный регистр отличается от верхнего. Но это не имеет значения для сортировки базового ASCII, а если вы собираетесь правильно выполнять сортировку Unicode, то подпрограммы для канонизации должны быть значительно изощреннее, чем `lc`.)

Функция `sort` часто применяется для сортировки хешей по значениям. Например, если хеш `%sales_amount` содержит записи объемов продаж по подразделениям, выполнение в программе сортировки поиска в хеше позволяет отсортировать ключи по соответствующим им значениям:

```
# сортировать по объемам продаж по подразделениям в порядке убывания
sub bysales { $sales_amount{$b} <=> $sales_amount{$a} }
for $dept (sort bysales keys %sale_amount) {
    print "$dept => $sales_amount{$dept}\n";
}
```

Можно осуществлять дополнительные уровни сортировки, каскадируя несколько сравнений с помощью операторов `||` или `or`. Эта конструкция работает прекрасно, поскольку операторы сравнения, что весьма удобно, возвращают `0` при эквивалентности, благодаря чему выполнение «проваливается» к следующему сравнению. В следующем примере ключи хеша сортируются сначала по связанным с ними объемам продаж, а затем по самим ключам (в случае, если два или более подразделений имеют одинаковый объем продаж):

```
sub by_sales_then_dept {
    $sales_amount{$b} <=> $sales_amount{$a}
       ||
    $a cmp $b
}

for $dept (sort by_sales_then_dept keys %sale_amount) {
```

```
    print "$dept => $sales_amount{$dept}\n";
}
```

Предположим, что @recs представляет собой массив ссылок на хеши, и каждый хеш содержит поля FIRSTNAME, LASTNAME, AGE, HEIGHT и SALARY. Следующий код сортирует записи о людях, перемещая вперед тех, кто богаче, затем выше, затем моложе, затем в алфавитном порядке имен:

```
sub prospects {
    $b->{SALARY}    <=>  $a->{SALARY}
        ||
    $b->{HEIGHT}    <=>  $a->{HEIGHT}
        ||
    $a->{AGE}       <=>  $b->{AGE}
        ||
    $a->{LASTNAME}  cmp  $b->{LASTNAME}
        ||
    $a->{FIRSTNAME} cmp  $b->{FIRSTNAME}
}

@sorted = sort prospects @recs;
```

Любая полезная информация, которую можно извлечь из $a и $b, может быть положена в основу сравнения в процедуре сортировки. Например, если строки текста должны быть отсортированы согласно определенным полям, для извлечения полей при сортировке можно использовать функцию split.

```
@sorted_lines = sort {
    @a_fields = split /:/, $a;      # поля, разделяемые двоеточиями
    @b_fields = split /:/, $b;
    $a_fields[3] <=> $b_fields[3]  # числовая сортировка по 4-му полю, затем
        ||
    $a_fields[0] cmp $b_fields[0]  # сортировка строк по 1-му полю, затем
        ||
    $b_fields[2] <=> $a_fields[2]  # числовая сортировка в порядке убывания
                                   # по 3-му полю
        ||
    ...                            # и т.д.
} @lines;
```

Однако поскольку sort выполняет программу сортировки много раз с различными комбинациями значений $a и $b, то в приведенном примере каждая строка будет повторно расщепляться чаще, чем это необходимо.

Чтобы избежать затрат на повторное выполнение таких действий, как расщепление строк для сравнения их полей, осуществите это действие по извлечению данных один раз перед сортировкой и сохраните полученную информацию. Вот как создаются анонимные массивы для хранения каждой строки вместе с результатами ее расщепления:

```
@temp = map { [$_, split /:/] } @lines;
```

После этого сортируются ссылки на массивы:

```
@temp = sort {
    @a_fields = @$a[1..$#$a];
    @b_fields = @$b[1..$#$b];
    $a_fields[3] <=> $b_fields[3]  # числовая сортировка по 4-му полю, затем
        ||
    $a_fields[0] cmp $b_fields[0]  # сортировка строк по 1-му полю, затем
        ||
    $b_fields[2] <=> $a_fields[2]  # числовая сортировка в порядке убывания
                                   # по 3-му полю
        ||
    ...                            # и т. д.
} @temp;
```

Теперь, когда отсортированы ссылки на массивы, из них можно извлечь исходные строки:

```
@sorted_lines = map { $_->[0] } @temp;
```

Объединяя все вместе, эта технология `map-sort-map`, которую часто называют преобразованием Шварца (Schwartzian Transform)[1], может быть выполнена в виде одной команды:

```
@sorted_lines = map { $_->[0] }
                sort {
                    @a_fields = @$a[1..#$a];
                    @b_fields = @$b[1..#$b];

                    $a_fields[3] <=> $b_fields[3]
                        ||
                    $a_fields[0] cmp $b_fields[0]
                        ||
                    $b_fields[2] <=> $a_fields[2]
                        ||
                    ...
                }
                map { [$_, split /:/] } @lines;
```

Не объявляйте `$a` и `$b` как лексические переменные (с помощью `my`). Они представляют собой глобальные переменные пакета (хотя на них не действуют обычные ограничения на глобальные переменные при использовании `use strict`). Однако необходимо гарантировать, что программа сортировки находится в том же пакете, либо квалифицировать `$a` и `$b` именем пакета вызывающей подпрограммы.

Ко всему этому добавим, что в версии 5.6 *можно* писать подпрограммы сортировки с обычным способом передачи аргументов (и, что не случайно,

---

[1] В честь ее автора – Рэндала Шварца. Описывается, например, на странице *http://members.home.net/andrew-johnson/perl/archit/msg00041.html*. – *Примеч. ред.*

использовать подпрограммы XS в качестве подпрограмм сортировки), при условии, что эта подпрограмма сортировки объявляется с прототипом ($$). А если сделаеть это, то можно на самом деле объявить $a и $b как лексические переменные:

```
sub numerically ($$) {
        my ($a, $b) = @_;
        $a <=> $b;
}
```

А когда-нибудь, когда будут реализованы полные прототипы, вы сможете просто сказать:

```
sub numerically ($a, $b) { $a <=> $b }
```

и тогда мы вернемся к тому, с чего начали. Более или менее.

splice

$@

```
splice ARRAY, OFFSET, LENGTH, LIST
splice ARRAY, OFFSET, LENGTH
splice ARRAY, OFFSET
splice ARRAY
```

Удаляет элементы, задаваемые *OFFSET* и *LENGTH*, из *ARRAY* и заменяет их элементами из *LIST*, если он задан. Если смещение *OFFSET* отрицательно, функция ведет обратный отсчет с конца массива, но если в результате окажется перед началом массива, то создается исключительная ситуация. В списковом контексте splice возвращает элементы, удаленные из массива. В скалярном контексте она возвращает последний удаленный элемент или undef, если такого не было. Если число новых элементов не равно числу прежних элементов, массив при необходимости расширяется или сокращается, и после выполнения функции элементы соответствующим образом изменяют свои места. Если длина *LENGTH* опущена, функция удаляет все до конца, начиная с *OFFSET*. Если смещение *OFFSET* опущено, массив после прочтения очищается. Имеют место следующие эквиваленты (в предположении, что $[равна 0):

Прямой метод	Эквивалент splice
push(@a, $x, $y)	splice(@a, @a, 0, $x, $y)
pop(@a)	splice(@a, -1)
shift(@a)	splice(@a, 0, 1)
unshift(@a, $x, $y)	splice(@a, 0, 0, $x, $y)
$a[$x] = $y	splice(@a, $x, 1, $y)
(@a, @a = ())	splice(@a)

Функцию splice удобно также применять для разрезания списка аргументов, переданных подпрограмме. Допустим, например, что перед списками передаются их длины:

```
sub list_eq {       # сравнить два списковых значения
    my @a = splice(@_, 0, shift);
    my @b = splice(@_, 0, shift);
    return 0 unless @a == @b;       # одинаковая длина?
    while (@a) {
        return 0 if pop(@a) ne pop(@b);
    }
    return 1;
}
if (list_eq($len, @foo[1..$len], scalar(@bar), @bar)) { ... }
```

Однако аккуратнее использовать для этого ссылки на массивы.

split

$_ T

```
split /PATTERN/, EXPR, LIMIT
split /PATTERN/, EXPR
split /PATTERN/
split
```

Эта функция просматривает строку, заданную *EXPR*, находит в ней разделители и расщепляет строку в список подстрок, возвращая в списковом контексте полученный список и в скалярном контексте число подстрок.[1] Разделители находятся путем многократного поиска по шаблону с использованием регулярного выражения, заданного в *PATTERN*, поэтому разделители могут иметь любой размер и не обязаны быть одной и той же строкой в каждом поиске. (Обычно разделители не возвращаются; исключения из этого правила излагаются ниже.) Если шаблон вовсе не удается отыскать в строке, то `split` возвращает исходную строку в качестве единственной подстроки. При нахождении одного соответствия возвращаются две подстроки и т. д. Можно задавать в шаблоне модификаторы регулярных выражений, например, `/PATTERN/i`, `/PATTERN/x` и т. д. В случае расщепления по шаблону `/^/` предполагается наличие модификатора `//m` .

Если число *LIMIT* задано и больше нуля, то функция расщепляет не более чем на заданное число полей (хотя может расщепить на меньшее, если разделителей окажется меньше). Если число *LIMIT* отрицательно, то считается, что была задана произвольно большая граница *LIMIT*. Если граница опущена или равна нулю, замыкающие нулевые поля удаляются из результата (о чем следует помнить тем, кто будет использовать `pop`). Если опущено *EXPR*, функция расщепляет строку `$_`. Если шаблон *PATTERN* тоже опущен или задан как пробел, `" "`, функция осуществляет расщепление по пробельным символам, `/\s+/`, пропустив сначала все ведущие пробельные символы.

Можно расщеплять строки любой длины:

---

[1] В скалярном контексте `split` записывает также результат в `@_`, но такое применение осуждается.

```
@chars  = split //,   $word;
@fields = split /:/,  $line;
@words  = split " ",  $paragraph;
@lines  = split /^/,  $buffer;
```

Шаблон, который может находить соответствие нулевой строке или чему-то более длинному, чем нулевая строка (например, шаблон, состоящий из одного символа, модифицированного * или ?), расщепляет значение *EXPR* на отдельные символы, если находит нулевую строку между символами; ненулевые соответствия пропускают символы найденного разделителя обычным образом. (Иными словами, шаблон не будет соответствовать в одной точке более одного раза, даже если найдено соответствие нулевой ширины.) Например:

```
print join ':', split / */, 'hi there';
```

дает результат «h:i:t:h:e:r:e». Пробел исчезает, поскольку он отыскивается как часть разделителя. В тривиальном случае нулевой шаблон // просто расщепляет строку на отдельные символы, и пробелы не исчезают. (При обычном поиске по шаблону // повторяет поиск по последнему успешно найденному шаблону, но в split шаблон представляет собой исключение из этого способа.)

Задание параметра *LIMIT* влечет расщепление только части строки:

```
($login, $passwd, $remainder) = split /:/, $_, 3;
```

Мы рекомендуем такое расщепление в список имен, которое позволит сделать код самодокументирующимся. (При проверке ошибок учитывайте, что $remainder окажется неопределенным, если в строке меньше трех полей.) Если осуществляется присвоение списку и пропущен *LIMIT*, Perl передает *LIMIT* на единицу больше, чем число переменных в списке, чтобы избежать лишней работы. Для приведенного выше расщепления *LIMIT* будет иметь по умолчанию значение 4, и $remainder получит только третье поле, а не все оставшиеся поля. В приложениях, критичных по времени выполнения, следует выделять не более полей, чем действительно требуется. (Коварство мощных языков в том, что временами они позволяют проявлять сильную глупость.)

Ранее мы говорили, что разделители не возвращаются, но если *PATTERN* содержит круглые скобки, то в результирующий список включаются подстроки, соответствующие каждой паре скобок, перемежаясь с обычно возвращаемыми полями. Вот простой пример:

```
split /([-,])/, "1-10,20";
```

который, создает списковое значение:

```
(1, '-', 10, ',', 20)
```

Если скобок больше, то поле возвращается для каждой пары, даже если каким-то парам не найдено соответствия, и тогда в этих позициях возвращаются неопределенные значения. Поэтому, если сказать:

```
split /(-)|(,)/, "1-10,20";
```

то будет получено значение:

```
(1, '-', undef, 10, undef, ',', 20)
```

Аргумент */PATTERN/* можно заменить выражением, которое будет задавать шаблоны, изменяющиеся во время выполнения программы. Как и для обычных шаблонов, чтобы выполнять компиляцию шаблона во время исполнения только один раз, используйте `/$variable/o`.

Особым случаем является задание в качестве выражения одного пробела (`" "`); в этом случае функция производит расщепление по пробельным символам, как `split` без аргументов. Поэтому с помощью `split(" ")` можно эмулировать режим работы *awk* по умолчанию. Напротив, `split(/ /)` даст столько нулевых полей в начале, сколько имеется ведущих пробелов. (Еще одним особым случаем является задание строки вместо регулярного выражения: она все равно будет интерпретирована как регулярное выражение.) Это свойство можно использовать для удаления из строки ведущих и замыкающих пробелов и замены встречающихся в середине серий пробельных символов на один пробел:

```
$string = join(' ', split(' ', $string));
```

В следующем примере заголовок сообщения RFC 822 расщепляется в хеш, содержащий `$head{Date}`, `$head{Subject}` и т. д. Используется прием присвоения хешу списка пар, основанный на том обстоятельстве, что разделители чередуются с разделяемыми полями. Для того чтобы вернуть часть каждого разделителя как часть возвращаемого спискового значения, указываются круглые скобки. Поскольку шаблон `split` гарантированно возвращает значения парами, т. к. содержит одну пару круглых скобок, то присвоение хешу гарантированно получает список, состоящий из пар ключ/значение, где каждый ключ является именем поля заголовка. (К сожалению, этот прием приводит к потере информации, если несколько строк имеют одинаковое ключевое поле, например строки Received-By. Увы...)

```
$header =~ s/\n\s+/ /g;       # Слить строки продолжения.
%head = ('FRONTSTUFF', split /^(\S*?):\s*/m, $header);
```

В следующем примере обрабатываются записи файла *passwd*(5) Unix. Можно опустить `chomp`, и тогда `$shell` будет иметь в конце символ перевода строки.

```
open PASSWD, '/etc/passwd';
while (<PASSWD>) {
    chomp;        # удалить замыкающие переводы строки
    ($login, $passwd, $uid, $gid, $gcos, $home, $shell) =
            split /:/;
    ...
}
```

Вот как можно обработать каждое слово каждой строки каждого входного файла, чтобы создать хеш с частотами слов:

```
while (<>) {
    foreach $word (split) {
        $count{$word}++;
    }
}
```

Действие, обратное split, осуществляет join (если не считать, что join может устанавливать между всеми полями только один разделитель). Для разбиения на части строки с полями фиксированной длины используйте unpack.

sprintf

sprintf *FORMAT*, *LIST*

Эта функция возвращает строку, отформатированную в соответствии с обычными соглашениями printf функции *sprintf* библиотеки C. Разъяснение общих принципов см. в *sprintf* (3) или *printf* (3) для вашей системы. Строка *FORMAT* содержит текст, в который встроены спецификаторы полей, заменяемых элементами *LIST*, по одному для каждого поля.

Perl осуществляет собственное форматирование sprintf: он эмулирует C-функцию *sprintf*, но не использует ее.[1] Вследствие этого любые нестандартные расширения, имеющиеся в локальной функции *sprintf* (3), не доступны в Perl.

Функция Perl sprintf разрешает общеизвестные преобразования, приведенные в табл. 29.4.

Таблица 29.4. Форматы для sprintf

Поле	Значение
%%	Знак процента
%c	Символ с заданным номером
%s	Строка
%d	Целое со знаком, в десятичном виде
%u	Целое без знака, в десятичном виде
%o	Целое без знака, в восьмеричном виде
%x	Целое без знака, в шестнадцатеричном виде
%e	Число с плавающей точкой, в научном формате
%f	Число с плавающей точкой, в фиксированном десятичном формате
%g	Число с плавающей точкой, в формате %e или %f

Кроме того, Perl разрешает следующие широко поддерживаемые преобразования:

[1] За исключением чисел с плавающей точкой, но даже тогда допускаются только стандартные модификаторы.

Поле	Значение
%X	Как %x, но с использованием букв на верхнем регистре
%E	Как %e, но с использованием «E» на верхнем регистре
%G	Как %g, но с использованием «E» на верхнем регистре (если применимо)
%b	Целое без знака, в двоичном виде
%p	Указатель (выводит адрес величины в шестнадцатеричном виде)
%n	Особый: *записывает* число выведенных до сих пор символов в следующую переменную в списке аргументов

Наконец, для обратной (мы подразумеваем именно «направленной назад») совместимости Perl разрешает следующие необязательные, но широко поддерживаемые преобразования:

Поле	Значение
%i	Синоним для %d
%D	Синоним для %ld
%U	Синоним для %lu
%O	Синоним для %lo
%F	Синоним для %f

Perl разрешает использовать следующие общеизвестные флаги между % и символом преобразования:

Флаг	Значение
пробел	Пробел перед положительным числом
+	Плюс перед положительным числом
-	Выравнивание по левому краю поля
0	Использовать нули, а не пробелы для выравнивания по правому краю поля
#	Использовать в качестве префикса ненулевых восьмеричных чисел "0", ненулевых шестнадцатеричных "0x"
число	Минимальная ширина поля
.число	«Точность»: количество цифр после десятичной точки в числах с плавающей точкой, максимальная длина для строки, минимальная длина для целого
l	Интерпретировать целое число как C-тип long или unsigned long
h	Интерпретировать целое число как C-тип short или unsigned short (если флаги не заданы, интерпретировать целое число как C-тип int или unsigned)

Есть также два специфических флага Perl:

Флаг	Значение
V	Интерпретировать целое число как стандартный тип целых Perl
v	Интерпретировать строку как вектор целых чисел, выводить как целые, разделяемые точками или произвольной строкой, полученной из списка аргументов, если перед флагом стоит *

Если «ваш» Perl понимает учетверенные слова («quads», 64-разрядные целые) вследствие того, что их поддерживает платформа или Perl был специально скомпилирован с поддержкой такой возможности, то символы `d u o x X b i D U O` выводят учетверенные слова, и этим символам могут предшествовать `ll`, `L` или `q`. Например, `%lld %16LX %qo`.

Если Perl понимает длинные числа двойной точности – «long doubles» (они должны поддерживаться платформой), то флагам `e f g E F G` могут предшествовать `ll` или `L`. Например `%llf %Lg`.

Там, где во флагах должно присутствовать число, вместо него можно использовать звездочку «`*`», и тогда Perl использует следующий элемент списка аргументов в качестве данного числа (т. е. как ширину поля или точность). Если полученная с помощью «`*`» ширина поля отрицательна, то результат будет таким же, как при использовании флага «`-`»: выравнивание по левому краю.

Флаг `v` удобно применять для вывода порядковых номеров символов в произвольных строках:

```
sprintf "version is v%vd\n", $^V;           # Версия Perl
sprintf "address is %vd\n", $addr;          # Адрес IPv4
sprintf "address is %*vX\n", ":", $addr;    # Адрес IPv6
sprintf "bits are %*vb\n", " ", $bits;      # случайные битовые строки
```

sqrt

$_ $@

```
sqrt EXPR
sqrt
```

Эта функция возвращает квадратный корень из *EXPR*. Другие корни, например кубические, можно получить при помощи оператора `**`, возводящего число в дробную степень. Не пытайтесь применять эти подходы к отрицательным числам, т. к. при этом возникает несколько более сложная задача (и создается исключительная ситуация). Но существует стандартный модуль, который может делать даже это:

```
use Math::Complex;
print sqrt(-2);    # выводит 1.4142135623731i
```

srand

```
srand EXPR
srand
```

Функция устанавливает случайное начальное число для оператора rand. Если выражение *EXPR* опущено, используется полуслучайное значение, предоставляемое ядром (если оно поддерживает устройство */dev/urandom*), или полученное, среди прочего, на основе текущего времени и ID процесса. Как правило, вообще нет необходимости вызывать srand, поскольку если не вызвать ее явно, она будет вызвана неявно при первом использовании оператора rand. Однако в версиях Perl до 5.004 это было не так, поэтому если сценарий должен выполняться со старыми версиями Perl, он должен вызывать srand.

Часто вызываемые программы (такие, как сценарии CGI), которые в качестве начального числа берут просто time ^ $$, могут стать жертвой того математического свойства, что a^b == (a+1)^(b+1) в одной трети случаев. Поэтому не делайте так. Лучше используйте:

```
srand( time() ^ ($$ + ($$ << 15)) );
```

Для криптографических задач понадобится что-нибудь гораздо более случайное, чем начальное число по умолчанию. В некоторых системах годится устройство */dev/random*. В иных случаях часто применяется контрольная сумма сжатого вывода одной или более программ быстро меняющегося состояния операционной системы. Например:

```
srand (time ^ $$ ^ unpack "%32L*", `ps wwaxl | gzip`);
```

Если вы этим особенно озабочены, ознакомьтесь с модулем Math::TrulyRandom в CPAN.

Не следует вызывать в программе srand несколько раз, кроме, конечно, тех случаев, когда вам точно известно, что и зачем вы делаете. Цель функции – дать начальное значение для функции rand, чтобы она могла создавать разные последовательности при каждом запуске программы. Сделайте это один раз в начале программы, иначе rand не даст вам случайные числа!

stat

```
stat FILEHANDLE
stat EXPR
stat
```

В скалярном контексте эта функция возвращает булево значение, указывающее на то, был ли успешным вызов. В списковом контексте она возвращает список из 13 элементов, содержащий статистику файла, открытого через указатель *FILEHANDLE* или поименованного в *EXPR*. Обычно она используется так:

```
($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,$size,
    $atime,$mtime,$ctime,$blksize,$blocks)
            = stat $filename;
```

Не все поля поддерживаются всеми типами файловых систем; неподдерживаемые поля возвращают 0. Значения полей приведены в табл. 29.5.

Таблица 29.5. Значения, возвращаемые stat

Номер	Поле	Значение
0	$dev	Номер устройства файловой системы
1	$ino	Номер inode
2	$mode	Режим файла (тип и разрешения)
3	$nlink	Число (жестких) ссылок на файл
4	$uid	Числовой ID пользователя владельца файла
5	$gid	Числовой ID назначенной группы файла
6	$rdev	Идентификатор устройства (только специальные файлы)
7	$size	Суммарный размер файла в байтах
8	$atime	Время последнего обращения к файлу в секундах с начала эпохи
9	$mtime	Время последней модификации файла в секундах с начала эпохи
10	$ctime	Время изменения inode (не время создания!) в секундах с начала эпохи
11	$blksize	Предпочтительный размер блока для ввода-вывода файловой системы
12	$blocks	Фактическое число выделенных блоков

Поля $dev и $ino в совокупности уникально идентифицируют файл в системе. $blksize и $blocks определены, скорее всего, только в файловых системах, производных от BSD. Поле $blocks (если определено) измеряется в блоках по 512 байт. Значение $blocks*512 может существенно отличаться от $size для файлов, содержащих невыделенные блоки, или «дыры», которые не учитываются в $blocks.

Если в stat передается специальный указатель файла, состоящий из символа подчеркивания, то фактически *stat*(2) не выполняется, а возвращается текущее содержимое структуры stat, полученной в результате последнего выполнения оператора тестирования файла, использовавшего stat, lstat или stat (например, -r, -w или -x).

Поскольку режим файла содержит его тип и права доступа, то чтобы увидеть реальные права, нужно замаскировать часть, содержащую тип файла, и вывести реальные права при помощи printf или sprintf со спецификатором «%o»:

```
$mode = (stat($filename))[2];
printf "Permissions are %04o\n", $mode & 07777;
```

Модуль File::stat предоставляет удобный механизм доступа по именам:

```
use File::stat;
$sb = stat($filename);
printf "File is %s, size is %s, perm %04o, mtime %s\n",
```

```
    $filename, $sb->size, $sb->mode & 07777,
    scalar localtime $sb->mtime;
```

Можно также импортировать символические определения различных разрядов режима из модуля Fcntl. Подробности рассмотрены в электронной документации.

Совет: для того чтобы получить только размер файла, обратитесь к оператору проверки файла -s, который прямо возвращает размер в байтах. Есть также проверки файла, возвращающие его возраст в днях.

study

```
study SCALAR
study
```

Эта функция тратит время на то, чтобы изучить *SCALAR*, предполагая проведение многократных поисков по шаблону, прежде чем строка будет модифицирована. При этом можно сберечь, а можно и не сберечь время, что зависит от природы и числа шаблонов поиска, и от распределения частот символов в строке, в которой осуществляется поиск, – возможно, понадобится сравнить время выполнения с участием study и без нее, чтобы узнать, что быстрее. Больше всего выигрывают от применения study циклы, в которых сканируется много коротких постоянных строк (в том числе постоянные части более сложных шаблонов). Если необходимо найти постоянные строки, привязанные к началу, study вообще не поможет, поскольку никакого сканирования не производится. Одновременно можно иметь только одну активную study; если начать изучение нового скаляра, то результаты для первого будут потеряны.

Работает study так. Создается связанный список для каждого символа строки, в которой будет производиться поиск, чтобы мы знали, например, где располагаются все символы «k». В каждой строке поиска на основывании некоторых статических таблиц частот, построенных с помощью C-программ и английского текста, выбирается символ, встречающийся реже всего. Исследуются только те места, где этот символ находится.

Вот, например, цикл, который вставляет элементы, порождающие индекс, перед каждой строкой, содержащей некоторый шаблон:

```
while (<>) {
    study;
    print ".IX foo\n"      if /\bfoo\b/;
    print ".IX bar\n"      if /\bbar\b/;
    print ".IX blurfl\n"   if /\bblurfl\b/;
    ...
    print;
}
```

При поиске /\bfoo\b/ будут рассматриваться только те места в $_, которые содержат «f», т. к. «f» встречается реже, чем «o». Это дает большой выигрыш, за исключением патологических случаев. Остается только невыяснен-

ным, больше ли при этом сбережется времени, чем было потрачено для построения связанного списка.

Если требуется искать строки, которые неизвестны до времени выполнения, можно создать цикл целиком как строку и выполнить для нее функцию `eval`, чтобы избежать постоянной перекомпиляции шаблонов. Если задать в `$/` ввод целых файлов как одной записи, это может оказаться очень быстрым, часто быстрее, чем применение специализированных программ вроде *fgrep*(1). Следующий код просматривает список файлов (`@files`) в поисках списка слов (`@words`) и выводит имена тех файлов, которые содержат нечувствительные к регистру соответствия:

```
$search = 'while (<>) { study;';
foreach $word (@words) {
    $search .= "++\$seen{\$ARGV} if /\\b$word\\b/i;\n";
}
$search .= "}";
@ARGV = @files;
undef $/;                # вводить файл целиком
eval $search;            # перехватывает исключительную ситуацию
die $@ if $@;            # в случае неудачи eval
$/ = "\n";               # восстановить обычное значение завершения ввода
foreach $file (sort keys(%seen)) {
    print "$file\n";
}
```

Сейчас, когда у нас есть оператор `qr//`, сложные `eval` этапа исполнения, которые мы видели выше, не столь необходимы. То же можно сделать с помощью следующего кода:

```
@pats = ();
foreach $word (@words) {
    push @pats, qr/\b${word}\b/i;
}
@ARGV = @files;
undef $/;                # вводить каждый файл целиком
while (<>) {
    for $pat (@pats) {
        $seen{$ARGV}++ if /$pat/;
    }
}
$/ = "\n";               # восстановить обычное значение завершения ввода
foreach $file (sort keys(%seen)) {
    print "$file\n";
}
```

sub

Именованные объявления:

```
sub NAME PROTO ATTRS
sub NAME ATTRS
```

```
sub NAME PROTO
sub NAME
```

Именованные определения:

```
sub NAME PROTO ATTRS BLOCK
sub NAME ATTRS BLOCK
sub NAME PROTO BLOCK
sub NAME BLOCK
```

Неименованные определения:

```
sub PROTO ATTRS BLOCK
sub ATTRS BLOCK
sub PROTO BLOCK
sub BLOCK
```

Синтаксис объявлений и определений подпрограмм выглядит сложным, но на практике он довольно прост. Все основывается на синтаксисе:

```
sub NAME PROTO ATTRS BLOCK
```

Все четыре поля не обязательны; единственное ограничение состоит в том, что поля, которые присутствуют, должны быть расположены в указанном порядке и что необходимо использовать хотя бы одно из полей *NAME* и *BLOCK*. Некоторое время мы не будем обращать внимание на *PROTO* и *ATTRS*; они являются просто модификаторами основного синтаксиса. А *NAME* и *BLOCK* являются важными параметрами, с которыми необходимо разобраться:

- Если указан только *NAME* без *BLOCK*, это объявление имени (а для того чтобы когда-либо вызвать эту подпрограмму, придется в дальнейшем дать определение, содержащее и *NAME* и *BLOCK*). Именованные объявления полезны, поскольку синтаксический анализатор обращается с именем особым образом, если знает, что это подпрограмма, определенная пользователем. Можно вызывать такую подпрограмму как функцию или как оператор – так же, как встроенные функции. Иногда их называют *предописаниями* (*forward declarations*).
- Если указаны и *NAME* и *BLOCK*, то это стандартное определение (definition) именованной подпрограммы (а также и объявление, если оно не было сделано раньше). Именованные определения полезны, потому что *BLOCK* связывает фактическое значение (тело подпрограммы) с объявлением. Это все, что мы имеем в виду, говоря, что это определение подпрограммы, а не просто объявление ее. Определение, однако, похоже на объявление, поскольку окружающий код не видит его, и оно не возвращает подставляемое значение, по которому можно ссылаться на подпрограмму.
- Если есть просто *BLOCK* без *NAME*, то это безымянное определение, т. е. анонимная подпрограмма. Поскольку у него нет имени, это вообще не объявление, а реальный оператор, который возвращает ссылку на тело анонимной подпрограммы на этапе исполнения. Это крайне удобно для работы с кодом как с данными и позволяет раскидывать случайные фрагменты

кода, чтобы использовать их как обратные вызовы, а может быть, даже как замыкания, если оператор определения sub ссылается на какие-либо внешние лексические переменные. Это значит, что различные вызовы одного и того же оператора sub будут вести бухгалтерию, необходимую для поддержания правильной «версии» каждой такой лексической переменной, которая видна в течение жизни замыкания, даже если изначальная область видимости лексической переменной разрушена.

В любом из этих трех случаев *PROTO* или *ATTRS* (а также они оба) могут присутствовать после *NAME* и/или перед *BLOCK*. Прототип представляет собой список символов в круглых скобках, которые сообщают анализатору, как обрабатывать аргументы функции. Атрибуты вводятся с помощью двоеточия и снабжают анализатор дополнительной информацией о функции. Вот типичное определение, которое включает все четыре поля:

```
sub numstrcmp ($$) : locked {
    my ($a, $b) = @_;
    return $a <=> $b || $a cmp $b;
}
```

Подробные сведения о списках атрибутов и работе с ними приведены в описании директивы attributes в главе 31. См. также главу 6 и раздел «Анонимные подпрограммы» главы 8.

substr

```
substr EXPR, OFFSET, LENGTH, REPLACEMENT
substr EXPR, OFFSET, LENGTH
substr EXPR, OFFSET
```

Эта функция извлекает подстроку из строки, заданной выражением *EXPR*, и возвращает ее. Подстрока извлекается, начиная с *OFFSET* символов от начала строки. (Примечание: если вы стали мудрить с $[, начало строки не находится в 0, но поскольку ничего такого вы не делали (или все-таки да?), это так.) Если смещение *OFFSET* отрицательно, подстрока начинается с указанного расстояния от конца строки. Если длина *LENGTH* опущена, то возвращается все до конца строки. Если *LENGTH* имеет отрицательное значение, то она рассчитывается так, чтобы оставить указанное число символов в конце строки. В противном случае *LENGTH* указывает длину извлекаемой подстроки, как вы, наверное, и предполагали.

Подстрока substr может выступать в качестве l-значения (того, чему можно присваивать значение), и в таком случае *EXPR* тоже должно быть l-значением. Если присваиваемое значение короче, чем подстрока, то строка сократится, а если длиннее, то строка соответствующим образом удлинится. Для сохранения прежней длины строки может потребоваться дополнить или обрубить ваше значение посредством sprintf или оператора x. При попытке присвоения невыделенной области за концом строки substr создает исключительную ситуацию.

Поместить перед строкой "Larry" текущее значение $_ можно так:

```
substr($var, 0, 0) = "Larry";
```

А так можно заменить первый символ $_ на "Moe":

```
substr($var, 0, 1) = "Moe";
```

И наконец, замена последнего символа $var на "Curly" выполняется следующим образом:

```
substr($var, -1) = "Curly";
```

Альтернативой применению substr в качестве левого значения является задание строки *REPLACEMENT* в качестве четвертого аргумента. Это позволяет заменять части *EXPR* и возвращать то, что в них находилось прежде, за одну операцию, как это делается в splice. Следующий пример тоже заменяет последний символ $var на "Curly" и помещает замененный символ в $oldstr:

```
$oldstr = substr($var, -1, 1, "Curly");
```

Не обязательно использовать substr как левое значение только в случае присвоения. Следующий пример заменяет все пробелы точками, но только в первых 10 символах строки:

```
substr($var, -10) =~ s/ /./g;
```

symlink

$! X T X U

```
symlink OLDNAME, NEWNAME
```

Функция создает новое имя файла, связанное символической ссылкой со старым именем файла. Возвращает значение «истина» в случае успеха и значение «ложь» в противном случае. В системах, не поддерживающих символические ссылки, она создает исключительную ситуацию на этапе исполнения. Чтобы проверить это, используйте eval для перехвата возможной ошибки:

```
$can_symlink = eval { symlink("","",); 1 };
```

Либо обратитесь к модулю Config. Будьте осторожны при передаче относительной символической ссылки, поскольку она будет интерпретироваться относительно расположения самой символической ссылки, а не относительно текущего рабочего каталога.

См. также link и readlink выше в этой главе.

syscall

$! X RO X T X U

```
syscall LIST
```

Вызывает системный вызов (т. е. syscall, а не команду оболочки), заданный в качестве первого элемента списка, и передает оставшиеся элементы как

аргументы системного вызова. (Доступ ко многим из этих вызовов теперь стал легче благодаря таким модулям, как POSIX.) Функция создает исключительную ситуацию, если *syscall*(2) не реализован.

Аргументы интерпретируются следующим образом: если данный аргумент представляет собой число, то он передается как целое число C. Если нет, передается указатель на строку. Программист отвечает за то, чтобы строка была достаточно длинна и могла принять любой результат, который может быть в нее записан; в противном случае его ожидает дамп памяти. Нельзя использовать строковый литерал (или другую строку, доступную только для чтения) в качестве аргумента syscall, поскольку Perl должен предполагать, что по любому указателю строки можно производить запись. Если целочисленные аргументы не являются литералами и никогда не интерпретировались в числовом контексте, то может потребоваться прибавить к ним 0, чтобы они выглядели как числа.

syscall возвращает то значение, которое было возвращено вызванным системным вызовом. В соответствии с соглашениями программирования на C, если системный вызов оказывается неудачным, то syscall возвращает -1 и устанавливает $! (номер ошибки). Допустимым возвращаемым значением некоторых системных вызовов в случае успеха является -1. Правильный способ обработки таких вызовов состоит в присвоении $!=0; перед вызовом и проверки значения $!, если syscall возвращает -1.

Не ко всем системным вызовам можно обращаться таким способом. Например, Perl поддерживает передачу до 14 аргументов в системный вызов, чего на практике как правило достаточно. Однако не все гладко с системными вызовами, возвращающими несколько значений. Возьмем syscall(&SYS_pipe): он возвращает номер файла на читающем конце создаваемого им канала. Не существует способа извлечь номер файла другого конца. Во избежание такого рода неприятностей можно обратиться к pipe. Для того чтобы обойти тупик в целом, напишите несколько XSUB (модулей внешних подпрограмм, диалект C) для непосредственного доступа к системным вызовам. Затем поместите свой новый модуль в CPAN и станьте ужасно знаменитым.

Следующая подпрограмма возвращает текущее время как число с плавающей точкой, а не целое число секунд, возвращаемое time. (Работает только на машинах, которые поддерживают системный вызов *gettimeofday*(2).)

```
sub finetime() {
    package main;   # для последующего require
    require 'syscall.ph';
    # установить размер буфера для двух 32-разрядных long...
    my $tv = pack("LL", ());
    syscall(&SYS_gettimeofday, $tv, undef) >= 0
        or die "gettimeofday: $!";
    my($seconds, $microseconds) = unpack("LL", $tv);
    return $seconds + ($microseconds / 1_000_000);
}
```

Предположим, что Perl не поддерживает системный вызов *setgroups*(2),[1] а ядро поддерживает. Все равно, можно добраться до него таким способом:

```
require 'syscall.ph';
syscall(&SYS_setgroups, scalar @newgids, pack("i*", @newgids))
        or die "setgroups: $!";
```

Как сказано в инструкциях по инсталляции Perl, для того чтобы *syscall.ph* существовал, может понадобиться запустить *h2ph*. На некоторых системах вместо этого может потребоваться шаблон `pack "II"`. Еще тревожнее, что `syscall` предполагает эквивалентность размеров C-типов `int`, `long` и `char*`. Постарайтесь не представлять себе `syscall` как изображение переносимости в миниатюре.

Более строгий подход к задачам определения времени с более высокой точностью реализован в модуле `Time::HiRes` из CPAN.

sysopen

```
sysopen FILEHANDLE, FILENAME, MODE, MASK
sysopen FILEHANDLE, FILENAME, MODE
```

Функция `sysopen` открывает файл, имя которого задано *FILENAME*, и связывает его с указателем файла *FILEHANDLE*. Если *FILEHANDLE* представляет собой выражение, то его значение используется как имя или ссылка на указатель файла. Если *FILEHANDLE* является переменной, значение которой не определено, то значение будет создано. Возвращается значение «истина», если вызов успешен, и «ложь» в противном случае.

Эта функция является прямым интерфейсом к системному вызову операционной системы *open*(2), за которым следует библиотечный вызов *fdopen*(3). По этой причине вам придется здесь немного прикинуться C-программистом. Возможные значения и биты флагов параметра *MODE* доступны через модуль `Fcntl`. Поскольку в различных системах поддерживаются различные флаги, не рассчитывайте, что все они доступны в вашей системе. За подробностями обращайтесь к странице руководства *open*(2) или ее локальному эквиваленту. Тем не менее, следующие флаги должны присутствовать в любой системе с достаточно стандартной библиотекой C:

Флаг	Значение
`O_RDONLY`	Только для чтения
`O_WRONLY`	Только для записи
`O_RDWR`	Чтение и запись
`O_CREAT`	Создать файл, если он не существует
`O_EXCL`	Ошибка, если файл уже существует

[1] Хотя через `$(` поддерживает.

(продолжение)

Флаг	Значение
O_APPEND	Дописывание к файлу
O_TRUNC	Усечение файла
O_NONBLOCK	Неблокирующий доступ

Однако доступно много других опций. Вот некоторые флаги, употребляемые реже:

Флаг	Значение
O_NDELAY	Старый синоним O_NONBLOCK
O_SYNC	Записывает блок, пока данные не будут физически записаны в устройство. Можно также встретить O_ASYNC, O_DSYNC и O_RSYNC
O_EXLOCK	flock с помощью LOCK_EX (только рекомендательная)
O_SHLOCK	flock с помощью LOCK_SH (только рекомендательная)
O_DIRECTORY	Отказ, если файл не является каталогом
O_NOFOLLOW	Отказ, если последняя часть пути представляет собой символическую ссылку
O_BINARY	Указатель binmode для систем Microsoft. Иногда бывает определен O_TEXT для противоположного режима
O_LARGEFILE	На некоторых системах требуется для файлов свыше 2 Гбайт
O_NOCTTY	Открытие файла терминала не делает последний управляющим терминалом процесса, если у вас его еще нет. Обычно не требуется

Флаг O_EXCL *не* служит для блокировки: в данном случае исключительность означает, что если файл уже существует, то вызов sysopen не завершается успехом.

Если файл, указанный *FILENAME*, не существует, а *MODE* содержит флаг O_CREAT, то sysopen создает файл, первоначальные права доступа к которому определяются аргументом *MASK* (или значением 0666, если он отсутствует), модифицируясь текущей маской umask вашего процесса. Это значение по умолчанию разумно: см. объяснение в описании umask.

Указатели файлов, открываемые с помощью open и sysopen, могут использоваться взаимозаменяемым образом. Нет необходимости в применении sysread со товарищи только потому, что файл оказался открытым с помощью sysopen, как этому не мешает и открытие его с помощью open. Каждый из них может делать то, чего не может другой. Обычный open может открывать каналы, ветвить процессы, устанавливать дисциплины, дублировать указатели файлов и преобразовывать номер дескриптора файла в указатель файла. Он также игнорирует ведущие и замыкающие пробельные символы в именах файлов и учитывает «-» в качестве специального имени файла. Но в отношении открытия фактических файлов sysopen может делать то же, что и open.

В следующих примерах приведены эквивалентные вызовы обеих функций. Для ясности мы опускаем проверки `or die $!`, но в своих программах всегда проверяйте возвращаемые значения. Мы ограничимся использованием только флагов, имеющихся практически в любых операционных системах. Все дело в том, чтобы следить за значениями, складываемыми с помощью поразрядного оператора | и передаваемыми в качестве аргумента *MODE*.

- Открыть файл для чтения:

```
open(FH, "<", $path);
sysopen(FH, $path, O_RDONLY);
```

- Открыть файл для записи, создавая при необходимости новый файл или усекая старый:

```
open(FH, ">", $path);
sysopen(FH, $path, O_WRONLY | O_TRUNC | O_CREAT);
```

- Открыть файл для дополнения, при необходимости создавая его:

```
open(FH, ">>", $path);
sysopen(FH, $path, O_WRONLY | O_APPEND | O_CREAT);
```

- Открыть для обновления файл, который должен уже существовать:

```
open(FH, "+<", $path);
sysopen(FH, $path, O_RDWR);
```

А вот что можно делать с помощью `sysopen` (но *не* обычного `open`):

- Открыть и создать файл для записи, который не должен предварительно существовать:

```
sysopen(FH, $path, O_WRONLY | O_EXCL | O_CREAT);
```

- Открыть файл для дополнения, который должен уже существовать:

```
sysopen(FH, $path, O_WRONLY | O_APPEND);
```

- Открыть файл для обновления, создавая при необходимости новый файл:

```
sysopen(FH, $path, O_RDWR | O_CREAT);
```

- Открыть файл для обновления, который не должен предварительно существовать:

```
sysopen(FH, $path, O_RDWR | O_EXCL | O_CREAT);
```

- Открыть файл только для записи без блокировки, но не создавать, если он не существует:

```
sysopen(FH, $path, O_WRONLY | O_NONBLOCK);
```

Модуль `FileHandle`, описываемый в главе 32, предоставляет группу объектно-ориентированных синонимов (плюс небольшое число новых функций) для открытия файлов. Можно вызывать соответствующие методы из `File-`

Handle[1] с любым указателем, созданным с помощью open, sysopen, pipe, socket или accept, даже если эти указатели были инициализированы без использования данного модуля.

sysread

$! $@ T X ARG X RO

```
sysread FILEHANDLE, SCALAR, LENGTH, OFFSET
sysread FILEHANDLE, SCALAR, LENGTH
```

Эта функция пытается прочесть *LENGTH* байт данных в переменную *SCALAR* из указанного *FILEHANDLE* с помощью системного вызова низкого уровня *read*(2). Функция возвращает число прочтенных байт или 0 при достижении EOF.[2] Функция sysread возвращает undef в случае ошибки. *SCALAR* будет расти или сокращаться в соответствии с фактически считанной длиной. Смещение *OFFSET*, если оно задано, указывает, в какое место строки начать помещать байты, поэтому можно производить чтение в середину строки, которой отводится роль буфера. Пример использования *OFFSET* см. в описании функции syswrite. Если *LENGTH* отрицательна или *OFFSET* указывает позицию за пределами строки, создается исключительная ситуация.

Следует быть готовым к решению возникающих проблем (таким, как прерываемые системные вызовы), которые обычно обрабатывает стандартный ввод-вывод. Поскольку функция обходит стандартный ввод-вывод, не смешивайте sysread с другими видами чтения, print, printf, write, seek, tell или eof на одном и том же указателе файла, если только вы не заинтересованы в сильнодействующем колдовстве (и/или головной боли). Кроме того, учитывайте, что при чтении файла, содержащего Unicode или другую многобайтовую кодировку, граница буфера может попасть на середину символа.

sysseek

$! X ARG

```
sysseek FILEHANDLE, POSITION, WHENCE
```

Эта функция устанавливает системную позицию *FILEHANDLE* с помощью системного вызова *lseek*(2). Она обходит стандартный ввод-вывод, поэтому смешивание ее с операциями чтения (отличными от sysread), print, write, seek, tell или eof может вызвать неприятности. *FILEHANDLE* может быть выражением, значение которого дает имя указателя файла. Значениями *WHENCE* являются 0 для установки новой позиции в *POSITION*, 1 для установки в текущую позицию плюс *POSITION* и 2 для установки в EOF плюс *POSITION* (обычно отрицательную). В качестве *WHENCE* могут выступать константы SEEK_SET, SEEK_CUR и SEEK_END из стандартных модулей IO::Seekable и POSIX — или, начиная с версии 5.6, из Fcntl, которые лучше переносимы и более удобны.

---

[1] Речь идет о методах IO::File или IO::Handle.

[2] Функции syseof нет, и это нормально, поскольку eof в любом случае не очень хорошо работает с файлами устройств (например, с терминалами). Используйте sysread и сравнивайте возвращаемое значение с 0, чтобы определить конец файла.

Функция возвращает новую позицию (или undef – в случае неудачи). Позиция ноль возвращается в виде особой строки "0 but true", которая может использоваться как число, не вызывая вывода предупреждений.

system

$! $? X T

```
system PATHNAME LIST
system LIST
```

Эта функция выполняет любую программу в системе и возвращает статус завершения этой программы, а не ее вывод. Для перехвата вывода команды используйте обратные штрихи или qx//. Функция system работает так же, как exec, за исключением того, что system сначала выполняет fork, а затем, после exec, ждет завершения выполняемой программы. Это значит, что она запускает указанную программу и возвращается по ее завершении, тогда как exec *заменяет* выполняемую программу новой и уже не возвращается, если замена прошла успешно.

Обработка аргументов различается в зависимости от их количества; она описывается в exec, в том числе – определение необходимости вызова оболочки и попытки обмана программы относительно ее имени указанием отдельного *PATHNAME*.

Поскольку system и обратные штрихи блокируют SIGINT и SIGQUIT, отправка одного из этих сигналов (например, от <Ctrl<+<C>) в выполняемую программу не прерывает вашей основной программы. Но другая программа, которую вы выполняете, *получит* сигнал. Проверяйте значение, возвращаемое system, чтобы знать, правильно ли завершилась запущенная вами программа.

```
@args = ("command", "arg1", "arg2");
system(@args) == 0
     or die "system @args failed: $?"
```

Возвращаемое значение представляет собой статус завершения программы в соответствии со значением, возвращенным системным вызовом *wait*(2). В традиционной семантике для получения реального кода завершения нужно разделить полученное значение на 256 (или сдвинуть вправо на 8 разрядов). Дело в том, что в младшем байте есть кое-что еще. (В действительности два кое-чего.) Младшие семь разрядов содержат номер сигнала, убившего процесс (если он был), а восьмой бит указывает, производил ли процесс дамп памяти. Можно проверить все варианты ошибок, включая сигналы и дамп памяти, рассмотрев $? ($CHILD_ERROR):

```
$exit_value  = $? >> 8;
$signal_num  = $? & 127;     # или 0x7f, или 0177, или 0b0111_1111
$dumped_core = $? & 128;     # или 0x80, или 0200, или 0b1000_0000
```

Если программа запущена через системную оболочку,[1] то, поскольку у вас только один аргумент и в нем содержатся метасимволы оболочки, обычные коды возврата подвергаются воздействию специфических особенностей и возможностей этой оболочки. Иными словами, в этом случае вы можете ока-

заться не в состоянии восстановить подробную информацию, как описано ранее.

syswrite

```
syswrite FILEHANDLE, SCALAR, LENGTH, OFFSET
syswrite FILEHANDLE, SCALAR, LENGTH
syswrite FILEHANDLE, SCALAR
```

Функция пытается записать *LENGTH* байт данных из переменной *SCALAR* в указанный *FILEHANDLE* с помощью системного вызова *write*(2). Возвращает число записанных байт (или `undef` – в случае ошибки). Смещение *OFFSET*, если оно задано, указывает, с какого места строки начать запись. (Это может понадобиться при использовании строки как буфера или необходимости восстановления при частичной записи.) Отрицательное смещение *OFFSET* указывает, что запись должна начаться, не доходя заданное число байт до конца строки. Если *SCALAR* пуста, допустимо только значение *OFFSET*, равное 0. Если *LENGTH* отрицательна или *OFFSET* указывает позицию вне строки, создается исключительная ситуация.

Чтобы скопировать данные из указателя файла `FROM` в указатель файла `TO`, можно сказать:

```
use Errno qw/EINTR/;
$blksize = (stat FROM)[11] || 16384;  # предпочтительный размер блока?
while ($len = sysread FROM, $buf, $blksize) {
    if (!defined $len) {
        next if $! == EINTR;
        die "System read error: $!\n";
    }
    $offset = 0;
    while ($len) {          # Обработка неполной записи.
        $written = syswrite TO, $buf, $len, $offset;
        die "System write error: $!\n" unless defined $written;
        $offset += $written;
        $len    -= $written;
    }
}
```

Следует быть готовым к решению возможных проблем, например, при частичной записи. Поскольку `syswrite` обходит стандартную библиотеку ввода-вывода C, не смешивайте ее вызовы (отличные от `sysread`), запись (например, `print`, `printf` или `write`) или другие функции стандартного ввода-вывода, такие как `seek`, `tell` или `eof`, если только вы не заинтересованы в сильнодействующем колдовстве.[1]

---

[1] По определению это */bin/sh* или то, что ей соответствует на вашей платформе, но не та оболочка, которая оказалась выполняемой пользователем в данный момент.

[1] Или головной боли.

tell

```
tell FILEHANDLE
tell
```

Эта функция возвращает текущую позицию в файле (в байтах, отсчет с нуля) для *FILEHANDLE*. Это значение обычно в какой-то момент будет передано на вход функции `seek`, чтобы вернуться к текущей позиции. *FILEHANDLE* может быть выражением, дающим имя текущего указателя файла, или ссылкой на объект указателя файла. Если *FILEHANDLE* опущен, то функция возвращает позицию в файле, из которого последний раз осуществлялось чтение. Позиции в файле имеют смысл только для обычных файлов. Для устройств, каналов и сокетов позиция в файле не существует.

Функции `systell` нет. Используйте для этой цели `sysseek(FH, 0, 1)`. См. в `seek` пример вызова `tell`.

telldir

```
telldir DIRHANDLE
```

Эта функция возвращает текущую позицию для программ `readdir` в указателе каталога *DIRHANDLE*. Это значение можно передать `seekdir` для доступа к конкретному месту в каталоге. Для этой функции действуют те же предостережения о возможном сжатии каталога, что и для соответствующей программы системной библиотеки. Функция может быть реализована не везде, где есть `readdir`. Даже если она есть, с возвращаемым значением нельзя производить расчеты. Это не вполне прозрачное значение, имеющее смысл только для `seekdir`.

tie

```
tie VARIABLE, CLASSNAME, LIST
```

Эта функция привязывает переменную к классу пакета, который обеспечивает реализацию этой переменной. *VARIABLE* представляет собой переменную (скаляр, массив или хеш) или typeglob (представляющий указатель файла), которую нужно связать. *CLASSNAME* является именем класса, реализующего объекты надлежащего типа.

Дополнительные аргументы передаются соответствующему конструктору класса и имеют значением одно из `TIESCALAR`, `TIEARRAY`, `TIEHASH` или `TIEHANDLE`. (Если соответствующий метод не найден, создается исключительная ситуация.) Обычно это те аргументы, которые могут быть переданы в функцию C *dbm_open*(3), но их значение зависит от пакета. Объект, возвращаемый конструктором, в свою очередь возвращается функцией `tie`, что может быть удобно, если надо обращаться к другим методам в *CLASSNAME*. (Доступ к объекту может быть также осуществлен через функцию `tied`.) Поэтому класс для связывания хеша с реализацией ISAM может предоставлять дополнитель-

ный метод для последовательного прохода множества ключей («S», входящее в ISAM, означает «sequential»), поскольку обычная реализация DBM может этого не делать.

Такие функции, как keys и values, могут возвращать огромные списки, если используются с такими большими объектами, как файлы DBM. Для их обхода более предпочтительным может оказаться применение функции each. Например:

```
use NDBM_File;
tie %ALIASES, "NDBM_File", "/etc/aliases", 1, 0
    or die "Невозможно открыть псевдонимы: $!\n";
while (($key,$val) = each %ALIASES) {
    print $key, ' = ', $val, "\n";
}
untie %ALIASES;
```

Класс, реализующий хеш, должен предоставлять следующие методы:

```
TIEHASH CLASS, LIST
FETCH SELF, KEY
STORE SELF, KEY, VALUE
DELETE SELF, KEY
CLEAR SELF
EXISTS SELF, KEY
FIRSTKEY SELF
NEXTKEY SELF, LASTKEY
DESTROY SELF
```

Класс, реализующий обычный массив, должен предоставлять следующие методы:

```
TIEARRAY CLASS, LIST
FETCH SELF, SUBSCRIPT
STORE SELF, SUBSCRIPT, VALUE
FETCHSIZE SELF
STORESIZE SELF, COUNT
CLEAR SELF
PUSH SELF, LIST
POP SELF
SHIFT SELF
UNSHIFT SELF, LIST
SPLICE SELF, OFFSET, LENGTH, LIST
EXTEND SELF, COUNT
DESTROY SELF
```

Класс, реализующий скаляр, должен предоставлять следующие методы:

```
TIESCALAR CLASS, LIST
FETCH SELF,
STORE SELF, VALUE
DESTROY SELF
```

Класс, реализующий указатель файла, должен предоставлять следующие методы:

```
TIEHANDLE CLASS, LIST
READ SELF, SCALAR, LENGTH, OFFSET
READLINE SELF
GETC SELF
WRITE SELF, SCALAR, LENGTH, OFFSET
PRINT SELF, LIST
PRINTF SELF, FORMAT, LIST
CLOSE SELF
DESTROY SELF
```

Не все указанные выше методы требуется реализовывать: модули Tie::Hash, Tie::Array, Tie::Scalar и Tie::Handle предоставляют базовые классы с разумными методами по умолчанию. Подробное обсуждение этих методов см. в главе 14 «Связанные переменные». В отличие от dbmopen, функция tie не станет загружать модуль через use или require — программист должен сделать это явно сам. Интересные реализации tie можно найти в модулях DB_File и Config.

tied

```
tied VARIABLE
```

Эта функция возвращает ссылку на объект, лежащий в основе скаляра, массива, хеша или typeglob, содержащихся в *VARIABLE* (то же значение, которое первоначально возвращалось вызовом tie, привязывавшим переменную к пакету). Она возвращает неопределенное значение, если *VARIABLE* не привязана к пакету. Поэтому, например, можно использовать:

```
ref tied %hash
```

чтобы определить, к какому пакету привязан ваш хеш. (Если вы забыли.)

time

```
time
```

Эта функция возвращает число секунд без учета високосных лет после «начала эпохи», обычно это 00:00:00 1 января 1970 года, UTC (Universal Coordinated Time).[1] Возвращаемое значение можно передать функциям gmtime и localtime для сравнения с временем модификации файла и доступа к файлу, возвращаемым stat, и в utime.

```
$start = time();
system("некая медленная команда");
```

[1] Не путать с «эпопеей», что относится к созданию Unix. (У других операционных систем могут быть другие эпохи, не говоря уже о других эпопеях.)

```
$end   = time();
if ($end - $start > 1) {
    print "Программа запущена: ", scalar localtime($start), "\n";
    print "Программа завершилась:   ", scalar localtime($end), "\n";
}
```

times

```
times
```

В списковом контексте эта функция возвращает список из четырех элементов, которые дают время в секундах (возможно, дробное), в течение которого пользователь и система занимали CPU, для данного процесса и его завершившихся дочерних процессов.

```
 ($user, $system, $cuser, $csystem) = times();
printf "Данный pid и его потомки использовали %.3f секунд\n",
            $user + $system + $cuser + $csystem;
```

В скалярном контексте функция возвращает просто время работы пользователя. Например, для измерения времени выполнения фрагмента кода Perl:

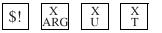
```
$start = times();
...
$end = times();
printf "Это заняло %.2f секунд использования CPU\n",
        $end - $start;
```

tr///

```
tr///
y///
```

Это оператор транслитерации (или трансляции), который подобен оператору `y///` в программе Unix *sed*, только лучший, по всеобщему скромному мнению. См. главу 5.

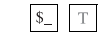
truncate



```
truncate FILEHANDLE, LENGTH
truncate EXPR, LENGTH
```

Эта функция усекает файл, открытый в *FILEHANDLE* или указанный по имени в *EXPR*, до указанной длины. Функция создает исключительную ситуацию, если *ftruncate*(2) или его эквивалент не реализованы в данной системе. (Всегда можно произвести усечение файла, копируя его начало, при наличии достаточного дискового пространства.) Функция возвращает значение «истина» в случае успеха и `undef` в противном случае.

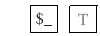
uc



```
uc EXPR
uc
```

Эта функция возвращает версию *EXPR* в верхнем регистре. Это внутренняя строка, которая реализует escape-последовательность `\U` в строках, заключенных в двойные кавычки. Perl постарается сделать «правильное дело», учитывая текущие национальные установки, но мы все еще работаем над взаимодействием функции с Unicode. См. последние достижения на странице руководства *perllocalle*. В любом случае при использовании Perl таблиц Unicode `uc` переводит символы в верхний регистр, а не в заглавный. О преобразовании в заглавный регистр см. `ucfirst` .

ucfirst



```
ucfirst EXPR
ucfirst
```

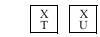
Возвращает версию *EXPR*, в которой первые символы переведены в заглавный регистр (в заглавном регистре «юникодского» языка), а остальные символы оставлены в покое. Это внутренняя функция, которая реализует escape-последовательность `\u` в строках, заключенных в двойные кавычки. Текущая установка `LC_CTYPE` может быть учтена, если выполняется `use locale` и ваши данные не напоминают Unicode, но никаких гарантий этого мы не даем.

Перевести начальный символ в заглавный регистр, а все остальное – в нижний можно так:

```
ucfirst lc $word
```

что эквивалентно `"\u\L$word"`.

umask



```
umask EXPR
umask
```

Эта функция устанавливает пользовательскую маску режима создаваемых процессом файлов и возвращает ее прежнее значение, используя системный вызов *umask*(2). Маска сообщает операционной системе, какие биты прав доступа запрещать при создании новых файлов, в том числе файлов, оказывающихся каталогами. Если *EXPR* опущено, функция просто возвращает текущую маску. Например, чтобы обеспечить разрешение битов «пользователя» и запрещение «других» битов, попробуйте что-нибудь вроде:

```
umask((umask() & 077) | 7);   # не изменять биты группы
```

Помните, что маска представляет собой число, обычно задаваемое в восьмеричном виде, а *не* строку восьмеричных цифр. Если же у вас есть только

строка, обратитесь к функции oct. Помните также, что биты маски являются дополнениями для обычных разрешений.

Права доступа Unix rwxr-x- - - представляются как три группы из трех бит или три восьмеричные цифры: 0750 (ведущий 0 указывает, что это восьмеричное число, и не считается за цифру). Поскольку биты маски пользователя перевернуты, она представляет выключенные биты разрешений. Значения прав доступа (или «режима»), передаваемые mkdir или sysopen, модифицируются маской пользователя, поэтому даже если потребовать от sysopen создать файл с режимом 0777, а маска пользователя равна 0022, то файл будет создан с правами доступа 0755. Если маска равна 0027 (группа не может писать; остальные не могут читать, писать или выполнять), то передача в sysopen *MASK*, равной 0666, создаст файл с режимом 0640 (поскольку 0666 & ~0027 равно 0640).

Вот некоторые советы: указывайте режим создания 0666 для обычных файлов (в sysopen) и 0777 для каталогов (в mkdir) и исполняемых файлов. Это дает пользователям свободу выбора: если им нужны защищенные файлы, они выбирают маску процесса 022, 027 или даже особенно антиобщественную маску 077. Программам едва ли вообще следует возлагать на пользователя принятие решения относительно политики. Исключением из этого правила служат программы, осуществляющие запись в файлы, которые должны быть закрытыми: почтовые файлы, cookies web-броузера, файлы *.rhosts* и т. д.

Если *umask*(2) не реализован в системе и вы пытаетесь ограничить свой *собственный* доступ (т. е. *EXPR* & 0700) > 0), будет создана исключительная ситуация на этапе исполнения. Если *umask*(2) не реализован и вы не пытаетесь ограничить свой собственный доступ, функция просто возвращает undef.

undef

```
undef EXPR
undef
```

undef – это имя, которым мы обозначаем абстракцию, известную как «неопределенное значение». Оно также удачно является именем функции, которая всегда возвращает неопределенное значение. Мы благополучно смешиваем одно с другим.

По некоторому совпадению функция undef может также явно сделать неопределенным объект, если передать его имя в качестве аргумента. Аргумент *EXPR*, если он задан, должен быть левым значением. Поэтому можно применять функцию только к скалярной величине, массиву или хешу целиком, имени подпрограммы (с префиксом &) или typeglob. Память, занимаемая объектом, освобождается и может повторно использоваться (хотя в большинстве операционных систем она не возвращается в систему). Функция undef может не сделать того, что от нее требуют, с большинством специальных переменных. Применение ее с доступной только для чтения переменной типа $1 создает исключительную ситуацию.

Функция undef является унарным оператором, а не списковым, поэтому сделать неопределенным одновременно можно только что-то одно. Вот несколько примеров использования undef как унарного оператора:

```
undef $foo;
undef $bar{'blurfl'};   # Не то же, что delete $bar{'blurfl'};
undef @ary;
undef %hash;
undef &mysub;
undef *xyz;             # разрушает $xyz, @xyz, %xyz, &xyz, etc.
```

Если аргумент не задан, undef используется в качестве своего значения:

```
select(undef, undef, undef, $naptime);
return (wantarray ? () : undef) if $they_blew_it;
return if $they_blew_it;  # то же
```

undef может выступать в качестве заполнителя в левой части присвоения списку, и тогда соответствующее значение в правой части просто отбрасывается. В другом качестве использовать undef как левое значение нельзя.

```
($a, $b, undef, $c) = &foo;       # Игнорировать третье возвращенное значение
```

Кроме того, не пытайтесь сравнивать что-либо с undef – получится не то, на что вы рассчитываете. Сравнение будет с 0 или нулевой строкой. Выяснить, определено ли значение, можно при помощи функции defined .

unlink

```
unlink LIST
unlink
```

Эта функция удаляет список файлов.[1] Функция возвращает число файлов, которые удалось удалить. Некоторые примеры:

```
$count = unlink 'a', 'b', 'c';
unlink @goners;
unlink glob("*.orig");
```

Функция unlink не станет удалять каталоги, если вы не являетесь суперпользователем и опция командной строки -U не передана Perl. Даже если эти условия выполнены, предупреждаем, что удаление каталога таким способом может нанести Серьезный Ущерб файловой системе. Используйте для этого rmdir.

---

[1] Фактически в файловой системе POSIX она удаляет записи каталога (имена файлов), которые ссылаются на реальные файлы. Поскольку ссылка на файл может иметься в нескольких каталогах, файл будет удален только тогда, когда будет удалена последняя ссылка на него.

Вот простая команда *rm* с очень простым контролем ошибок:

```
#!/usr/bin/perl
@cannot = grep {not unlink} @ARGV;
die "$0: could not unlink @cannot\n" if @cannot;
```

unpack


```
unpack TEMPLATE, EXPR
```

Эта функция оказывает действие, обратное pack: она раскрывает строку (*EXPR*), представляющую структуру данных, в список значений согласно шаблону *TEMPLATE* и возвращает эти значения. В скалярном контексте она может применяться для распаковки одного значения. *TEMPLATE* имеет формат, сходный с тем, который есть в функции pack, – он задает порядок и тип значений, которые должны быть распакованы. Подробное описание *TEMPLATE* приведено в pack. Недопустимый элемент в *TEMPLATE* либо попытка выйти за пределы строки в форматах x, X или @ создает исключительную ситуацию.

Строка разбивается на участки, описываемые *TEMPLATE*. Каждый участок отдельно преобразуется в значение. Обычно байты строки либо получены с помощью pack, либо представляют некую структуру C.

Если счетчик повторений поля больше, чем позволяет остаток входной строки, счетчик автоматически уменьшается. (Однако обычно в таких случаях в качестве счетчика повторений следует использовать *.) Если входная строка длиннее, чем описано в *TEMPLATE*, оставшаяся часть строки игнорируется.

Функция unpack может быть полезна и при работе с обычными текстовыми файлами, не обязательно двоичными. Предположим, что у вас есть файл данных с такими записями:

```
1986 Ender's Game           Orson Scott Card
1985 Neuromancer            William Gibson
1984 Startide Rising        David Brin
1983 Foundation's Edge      Isaac Asimov
1982 Downbelow Station      C. J. Cherryh
1981 The Snow Queen         Joan D. Vinge
```

Применять split для выделения полей нельзя, поскольку нет особого разделителя полей, а определяются поля по их смещению в записи. Поэтому, хотя это записи обычного текста, фиксированный формат позволяет разобрать их с помощью unpack:

```
while (<>) {
        ($year, $title, $author) = unpack("A4 x A23 A*", $_);
        print "$author won ${year}'s Hugo for $title.\n";
}
```

(Мы написали ${year}'s, т. к. Perl рассматривал бы $year's как $year::s.)

Вот полный текст программы *uudecode*:

```
#!/usr/bin/perl
$_ = <> until ($mode,$file) = /^begin\s*(\d*)\s*(\S*)/;
open(OUT,"> $file") if $file ne "";
while (<>) {
    last if /^end/;
    next if /[a-z]/;
    next unless int((((ord() - 32) & 077) + 2) / 3) ==
                int(length() / 4);
    print OUT unpack "u", $_;
}
chmod oct($mode), $file;
```

Кроме полей, допустимых в `pack`, можно задавать для поля префикс *%число*, что создает простую контрольную сумму объекта с указанным количеством разрядов, а не сам объект. По умолчанию это 16-разрядная сумма. Контрольная сумма создается путем сложения числовых значений распакованных величин (для строковых полей берется сумма `ord($char)`, а для битовых полей сумма нулей и единиц). Например, следующий код рассчитывает то же число, что и программа SysV *sum*(1):

```
undef $/;
$checksum = unpack ("%32C*", <>) % 65535;
```

Следующий код эффективно подсчитывает число установленных в единицу разрядов в битовой строке:

```
$setbits = unpack "%32b*", $selectmask;
```

Вот простой декодер BASE64:

```
while (<>) {
    tr#A-Za-z0-9+/##cd;                  # удалить символы не-base64
    tr#A-Za-z0-9+/# -_#;                 # преобразовать в формат uuencode
    $len = pack("c", 32 + 0.75*length);  # вычислить байт длины
    print unpack("u", $len . $_);        # uudecode и вывод
}
```

unshift

```
unshift ARRAY, LIST
```

Эта функция выполняет действие, противоположное `shift`. (Или противоположное `push`, в зависимости от того, как на это посмотреть.) Она добавляет список *LIST* к началу массива и возвращает новое число элементов в массиве:

```
unshift @ARGV, '-e', $cmd unless $ARGV[0] =~ /^-/;
```

Обратите внимание, что *LIST* добавляется целиком, а не по одному элементу, поэтому добавляемые элементы сохраняют свой порядок. Для обращения порядка используйте `reverse`.

untie

```
untie VARIABLE
```

Разрывает связь между переменной или typeglob, содержащимися в *VARIABLE*, и пакетом, к которому они привязаны. См. tie и всю главу 14, особенно раздел «Тонкая ловушка при отвязывании».

use

$! $@

```
use MODULE VERSION LIST
use MODULE VERSION ()
use MODULE VERSION
use MODULE LIST
use MODULE ()
use MODULE
use VERSION
```

Объявление use загружает модуль, если он не был загружен ранее, и импортирует в текущий пакет подпрограммы и переменные из указанного модуля. (Говоря формальным языком, оно импортирует в текущий пакет некоторую семантику из указанного модуля, обычно создавая в вашем пакете псевдонимы для некоторых имен подпрограмм и переменных.) Большинство объявлений use выглядит так:

```
use MODULE LIST;
```

Это в точности эквивалентно такому высказыванию:

```
BEGIN { require MODULE; import MODULE LIST; }
```

BEGIN заставляет require и import происходить во время компиляции. require обеспечивает загрузку модуля в память, если она еще не произведена. import не является встроенной функцией – это обычный вызов метода класса в пакете с именем *MODULE*, сообщающий этому модулю о необходимости загрузить список характеристик в текущий пакет. Модуль может реализовать свой метод импорта любым способом, хотя большинство модулей получают его путем наследования от класса Exporter, определенного в модуле Exporter. Дополнительные сведения см. в главе 11 «Модули» и в модуле Exporter. Если метод import не найден, вызов «молча» пропускается.

Для того чтобы предотвратить изменение своего пространства имен, явно задайте пустой список:

```
use MODULE ();
```

Это в точности эквивалентно следующему коду:

```
BEGIN { require MODULE; }
```

Если первый аргумент use является номером версии, например 5.6.2, то выполняемая в данный момент версия Perl должна быть не ниже указанной.

Если текущая версия Perl меньше *VERSION*, то выводится сообщение об ошибке и происходит немедленное завершение Perl. Это полезно для проверки текущей версии Perl перед загрузкой библиотечных модулей, зависящих от более новых версий, поскольку иногда нам приходится «ломать» ошибочные функции старых версий Perl. (Мы стараемся ломать не больше того, что необходимо. На самом деле мы часто стараемся ломать меньше, чем необходимо.)

Если говорить о том, чтобы не ломать, то нужно отметить, что Perl по-прежнему принимает старые номера версий в формате:

```
use 5.005_03;
```

Однако для лучшего согласования с промышленными стандартами Perl 5.6 теперь принимает (а мы предпочитаем видеть) трехэлементный формат:

```
use 5.6.0;   # Версия 5, подверсия 6, уровень патчей 0.
```

Если аргумент *VERSION* следует за *MODULE*, то `use` вызовет метод `VERSION` в классе *MODULE* с заданной *VERSION* в качестве аргумента. Обратите внимание, что после *VERSION* нет запятой! Метод `VERSION` по умолчанию, наследуемый от класса `UNIVERSAL`, создает исключительную ситуацию (croaks), если указанная версия превосходит значение переменной `$Module::VERSION`.

См. список стандартных модулей в главе 32.

Поскольку `use` обеспечивает широко открытый интерфейс, директивы (директивы компилятора) тоже реализуются посредством модулей. Вот примеры реализованных в настоящее время директив:

```
use autouse 'Carp' => qw(carp croak);
use bytes;
use constant PI => 4 * atan2(1,1);
use diagnostics;
use integer;
use lib '/opt/projects/spectre/lib';
use locale;
use sigtrap qw(die INT QUIT);
use strict  qw(subs vars refs);
use warnings "deprecated";
```

Многие из таких директивных модулей импортируют семантику в текущую область лексической видимости. (Это их отличие от обычных модулей, которые импортируют символы только в текущий пакет, имеющий мало отношения к текущей лексической области видимости помимо того, что текущая лексическая область видимости компилируется с учетом этого пакета. Это значит... а, не имеет значения, посмотрите главу 11.)

Имеется соответствующее объявление `no`, которое «деимпортирует» значения, первоначально импортированные `use`, ставшие ненужными:

```
no integer;
no strict 'refs';
```

```
no utf8;
no warnings "unsafe";
```

См. список стандартных директив в главе 31.

utime

```
utime LIST
```

Эта функция изменяет время обращения к файлу и время модификации файла для всех файлов, указанных в списке. Первые два элемента списка должны задавать время обращения и модификации в числовом виде и в указанном порядке. Функция возвращает число успешно измененных файлов. Время изменения каждого файла в описателе (inode) устанавливается равным текущему времени. Вот пример команды *touch*, которая устанавливает дату модификации файла (в предположении, что вы его владелец) примерно на месяц вперед:

```
#!/usr/bin/perl
# montouch - датировать файлы: сейчас + 1 месяц
$day = 24 * 60 * 60;            # 24 часа секунд
$later = time() + 30 * $day;    # 30 дней - это, примерно, месяц
utime $later, $later, @ARGV;
```

а вот более сложная *touch*-образная команда с некоторой проверкой ошибок:

```
#!/usr/bin/perl
# montouch - датировать файлы: сейчас + 1 месяц
$later = time() + 30 * 24 * 60 * 60;
@cannot = grep {not utime $later, $later, $_} @ARGV;
die "$0: Could not touch @cannot.\n" if @cannot;
```

Чтобы прочесть время для существующих файлов, используйте `stat`, а затем пропустите соответствующие поля через `localtime` или `gmtime` перед выводом.

values

```
values HASH
```

Эта функция возвращает список, состоящий из всех значений в указанном хеше *HASH*. Значения возвращаются в порядке, кажущемся случайным, но это тот же порядок, который создают функции `keys` и `each` для того же хеша. Как ни странно, для сортировки хеша по значениям обычно требуется функция `keys`, поэтому за примером обратитесь к описанию `keys`.

С помощью этой функции можно модифицировать значения в хеше, потому что возвращаемый список содержит не копии значений, а их псевдонимы. (В ранних версиях для этого требовался срез хеша.)

```
for (@hash{keys %hash}) { s/foo/bar/g }   # по-старому
for (values %hash)      { s/foo/bar/g }   # теперь это изменяет значения
```

Применение values с хешем, который связан с огромным файлом DBM, создаст большой список, а в результате будет получен большой процесс. Предпочтительным может оказаться использование здесь функции each, которая осуществляет проход через все записи хеша, не создавая из них при этом гигантского списка.

vec

```
vec EXPR, OFFSET, BITS
```

Функция vec обеспечивает компактное хранение списков беззнаковых целых. Эти целые плотно, насколько это возможно, упакованы в обычной строке Perl. Строка в *EXPR* рассматривается как строка битов, составленная из произвольного числа элементов, зависящего от длины строки.

OFFSET указывает индекс конкретного элемента, который нас интересует. Синтаксис чтения и записи элемента одинаков, поскольку vec записывает или возвращает значение элемента в зависимости от того, в каком контексте она используется – левого значения или правого значения.

BITS задает ширину каждого элемента в битах и должно быть степенью двух: 1, 2, 4, 8, 16 или 32 (на некоторых платформах также 64). (В случае применения других значений создается исключительная ситуация.) Поэтому каждый элемент может содержать целое в диапазоне 0..(2**BITS)-1. Для меньших размеров в каждый байт упаковывается столько элементов, сколько можно. Когда *BITS* равно 1, в байте находятся 8 элементов. Когда *BITS* равно 2 – четыре. Когда *BITS* равно 4, в байте два элемента (обычно называемые полубайтами или «ниблами» (nybbles)). И так далее. Целые, занимающие больше одного байта, хранятся в тупоконечом порядке.

Список целых чисел без знака можно хранить в одной скалярной переменной, присвоив их отдельно функции vec. (Если *EXPR* не является левым значением, создается исключительная ситуация.) В следующем примере все элементы имеют ширину 4 бита:

```
$bitstring = "";
$offset = 0;
foreach $num (0, 5, 5, 6, 2, 7, 12, 6) {
    vec($bitstring, $offset++, 4) = $num;
}
```

Если запись производится дальше конца строки, то Perl сначала расширяет строку необходимым количеством нулевых байтов.

Векторы, хранимые в скалярной переменной, можно последовательно извлекать, задавая правильное смещение *OFFSET*.

```
$num_elements = length($bitstring)*2;  # 2 элемента в байте
foreach $offset (0 .. $num_elements-1) {
    print vec($bitstring, $offset, 4), "\n";
}
```

Если выбираемый элемент находится за концом строки, возвращается значение 0.

Строки, создаваемые vec, можно также обрабатывать логическими операторами |, &, ^ и ~. Эти операторы считают, что нужна операция над битовыми строками, если оба операнда являются строками. См. соответствующие примеры в главе 3 «Унарные и бинарные операторы» в разделе «Поразрядные операторы».

Если *BITS* == 1, может быть создана битовая строка для записи серий битов в один скаляр. Порядок таков, что vec($bitstring,0,1) гарантированно попадает в младший бит первого байта строки.

```
@bits = (0,0,1,0, 1,0,1,0, 1,1,0,0, 0,0,1,0);
$bitstring = "";
$offset = 0;
foreach $bit (@bits) {
    vec($bitstring, $offset++, 1) = $bit;
}
print "$bitstring\n";        # "TC", т.е. '0x54', '0x43'
```

Битовая строка может транслироваться в строку из 1 и 0 и обратно путем передачи шаблона "b*" в pack или unpack. Можно также использовать pack с шаблоном "b*" для создания битовой строки из строки, состоящей из 1 и 0. Порядок совместим с тем, который предполагает vec.

```
$bitstring = pack "b*", join('', @bits);
print "$bitstring\n";   # "TC", то же, что раньше
```

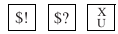
При помощи unpack можно организовать извлечение списка 0 и 1 из битовой строки.

```
@bits = split(//, unpack("b*", $bitstring));
print "@bits\n";        # 0 0 1 0 1 0 1 0 1 1 0 0 0 0 1 0
```

Если известна точная длина строки в битах, то ее можно задать вместо "*".

Дополнительные примеры использования битовых строк, создаваемых посредством vec можно найти в select. См. в pack и unpack примеры работы с двоичными данными более высокого уровня.

wait



```
wait
```

Эта функция ждет завершения дочернего процесса и возвращает PID завершенного процесса или -1, если дочерних процессов нет (в некоторых системах – если дочерние процессы убираются автоматически). Статус возвращается в $?, как описано в system. Если у вас образуются дочерние процессы-зомби, то следует вызывать эту функцию или waitpid.

Если вы ожидали найти дочерний процесс и не нашли его с помощью wait, то, возможно, вы вызывали system, закрытие канала или обратные штрихи между fork и wait. Эти конструкции тоже выполняют *wait*(2) и могли убрать дочерний процесс. Во избежание этого применяйте waitpid.

waitpid

$! $? X U

```
waitpid PID, FLAGS
```

Эта функция ждет завершения конкретного дочернего процесса и возвращает PID завершенного процесса или -1, если дочерних процессов нет, или 0, если флаги *FLAGS* задают не блокирующий вызов, а процесс еще не завершен. Статус завершенного процесса возвращается в $?, как описано в system. Чтобы иметь правильные значения флагов, необходимо импортировать группу тегов импорта ":sys_wait_h" из модуля POSIX. Вот пример, осуществляющий неблокирующее ожидание для всех незавершенных процессов-зомби.

```
use POSIX ":sys_wait_h";
do {
    $kid = waitpid(-1,&WNOHANG);
} until $kid == -1;
```

В системах, где не реализован ни один из системных вызовов *waitpid*(2) и *wait4*(2), флаги можно задать только как 0. Иными словами, вы можете ждать в них конкретный *PID*, но не можете делать это в неблокирующем режиме.

В некоторых системах возврат значения -1 может означать, что дочерние процессы автоматически убираются, т. к. пользователь установил $SIG{CHLD} = 'IGNORE'.

wantarray

```
wantarray
```

Эта функция возвращает значение «истина», если контекст выполняемой в данное время подпрограммы ищет списковое значение, и «ложь» в противном случае. Функция возвращает определенное ложное значение (""), если вызывающий контекст ищет скаляр, и неопределенное ложное значение (undef), если вызывающий контекст ничего не ищет; т. е. если функция находится в пустом контексте.

Вот примеры типичного использования:

```
return unless defined wantarray;    # больше не надо трудиться
my @a = complex_calculation();
return wantarray ? @a : \@a;
```

См. также caller. Эта функция в действительности должна была быть названа "wantlist", но мы дали ей имя еще тогда, когда списковый контекст назывался контекстом массива.

warn

$!

```
warn LIST
warn
```

Генерирует сообщение об ошибке, выводя *LIST* на STDERR как die, но не пытается завершить работу или создать исключительную ситуацию. Например:

```
warn "Отладка включена" if $debug;
```

Если список *LIST* пуст, а $@ уже содержит значение (обычно от предыдущего eval), то вслед за $@ на STDERR выводится строка «\t...caught». (Это аналогично тому, как die распространяет ошибки, за исключением того, что warn не распространяет (не создает повторно) исключительную ситуацию.) Если передана пустая строка сообщений, выводится сообщение "Warning: Something's wrong" (что-то случилось).

Как и в die, если передаваемые строки не заканчиваются символом новой строки, автоматически добавляются сведения о файле и номере строки. Функция warn не связана с опцией командной строки Perl –w, но может применяться совместно с ней, например, когда требуется эмулировать встроенную функцию:

```
warn "Something wicked\n" if $^W;
```

Сообщение не выводится, если установлен обработчик $SIG{_ _WARN_ _}. На обработчике лежит обязанность поступить с сообщением так, как он считает приемлемым. Например, можно поднять простое предупреждение до положения исключительной ситуации:

```
local $SIG{_ _WARN_ _} = sub {
    my $msg = shift;
    die $msg if $msg =~ /isn't numeric/;
};
```

Большинство обработчиков должно поэтому принять меры к тому, чтобы вывести предупреждения, к действиям с которыми они не готовы, путем повторного вызова warn в обработчике. Это совершенно безопасно и не породит бесконечный цикл, потому что ловушки _ _WARN_ _ не вызываются из ловушек _ _WARN_ _. Такой режим несколько отличается от действия обработчиков $SIG{_ _DIE_ _} (которые не подавляют текст ошибки, но могут повторно вызвать die, чтобы изменить его).

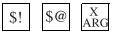
Обработчик _ _WARN_ _ предоставляет мощный способ заглушить все предупреждения, даже так называемые обязательные. Иногда требуется «завернуть» это в блок BEGIN{} для выполнения на этапе компиляции:

```
# выкинуть *все* предупреждения этапа компиляции
BEGIN { $SIG{_ _WARN_ _} = sub { warn $_[0] if $DOWARN } }
my $foo = 10;
my $foo = 20;          # нет предупреждения о дублировании my $foo,
```

```
                                # но вы сами этого хотели!
# до сих пор никаких предупреждений этапа компиляции или исполнения
$DOWARN = 1;                    # *не* встроенная переменная
# с этого места предупреждения этапа исполнения включены
warn "\$foo is alive and $foo!";      # выводится
```

См. в описании директивы `use warnings` об управлении предупреждениями с лексической областью видимости. Другие способы вывода предупредительных сообщений реализованы в функциях `carp` и `cluck` модуля `Carp`.

write



```
write FILEHANDLE
write
```

Эта функция выводит отформатированную запись (возможно, из нескольких строк) по заданному указателю файла, используя формат, связанный с этим указателем, – см. раздел «Переменные формата» главы 7. По умолчанию формат, связанный с указателем файла, имеет такое же, как у него, имя. Однако формат для указателя файла можно изменить, модифицировав переменную `$~` после выполнения `select` для этого указателя:

```
$old_fh = select(HANDLE);
$~ = "NEWNAME";
select($old_fh);
```

или сказав:

```
use IO::Handle;
HANDLE->format_name("NEWNAME");
```

Поскольку форматы помещаются в пространство имен пакета, может потребоваться полностью квалифицировать имя формата, если объявление `format` находится в другом пакете:

```
$~ = "OtherPack::NEWNAME";
```

Обработка верхнего колонтитула страницы производится автоматически: если на текущей странице недостаточно места для форматированной записи, страница перемещается выводом символа подачи страницы, для заголовка новой страницы используется специальный формат верхнего колонтитула страницы, а затем выводится запись. Число строк, оставшихся на текущей странице, находится в переменной `$-`, которую можно установить в 0, чтобы форсировать вывод новой страницы при следующем вызове `write`. (Может потребоваться сначала выполнить `select` для указателя файла.) По умолчанию именем формата верхнего колонтитула является имя указателя файла с добавлением `"_TOP"`, но формат для указателя файла можно изменить установкой переменной `$^` после выполнения `select` для этого указателя файла, либо сказав:

```
use IO::Handle;
HANDLE->format_top_name("NEWNAME_TOP");
```

Если *FILEHANDLE* не задан, то вывод производится в текущий указатель выходного файла, который вначале имеет значение STDOUT, но может быть изменен посредством оператора select с одним аргументом. Если *FILEHANDLE* представляет собой выражение, то на этапе исполнения выражение вычисляется для получения фактического *FILEHANDLE*.

Если указанный формат или формат верхнего колонтитула не существует, создается исключительная ситуация.

Функция write *не* является противоположной для read. К несчастью. Для простого вывода строк воспользуйтесь функцией print. Если вы открыли это описание в поисках обхода стандартного ввода-вывода, то обратитесь к функции syswrite.

y//

```
y///
```

Оператор транслитерации (исторически называемый также трансляцией), известный еще как tr///. См. главу 5.

30

Стандартная библиотека Perl

Стандартный дистрибутив Perl содержит значительно больше, чем только исполняемый файл *perl*, который выполняет ваши сценарии. В него также входят сотни модулей с кодом, который можно использовать повторно. Стандартные модули доступны всюду, поэтому программу, обращающуюся к ним, можно запускать везде, где установлен Perl, без дополнительных шагов при установке.

Библиотековедение

Прежде чем перечислить эти модули в последующих главах, приведем беглый обзор терминологии, которой мы пользуемся.

namespace

Пространство имен (*namespace*) представляет собой место, предназначенное для того, чтобы не путать имена, хранимые в нем, с именами в других пространствах имен. В результате остается меньшая проблема – не путать сами пространства имен. Есть два способа избежать смешения имен пространств между собой: дать им уникальные имена или уникальное местоположение. Perl позволяет сделать и то и другое: именованные пространства имен называются пакетами, а неименованные пространства имен называются областями лексической видимости. Поскольку области лексической видимости не могут быть больше чем файл и поскольку стандартные модули имеют размер файла (как минимум), отсюда следует, что все интерфейсы модулей должны использовать именованные пространства имен (пакеты), если с ними должен работать кто-то вне файла.

package

Пакет (*package*) – это стандартный механизм Perl для объявления именованного пространства имен. Он обеспечивает простой способ группировки связанных функций и переменных. Точно так же, как два каталога могут содержать (различные) файлы с именем *fred*, две различные части программы Perl могут содержать каждая свою переменную `$fred` или функцию `&fred`. Даже если этим переменным или функциям даны, казалось бы, одинаковые имена, эти имена находятся в различных пространствах имен, определяемых объявлениями пакетов. Имена пакетов служат для идентификации модулей и классов, как описано в главе 11 «Модули» и в главе 12 «Объекты».

library

Термин *библиотека* (*library*), к несчастью, довольно перегружен в культуре Perl. В настоящее время этим термином мы обычно обозначаем все множество модулей Perl, установленных на данной машине.

Исторически библиотека Perl была также файлом, содержащим набор подпрограмм, служащих некоей общей цели. Такой файл часто имеет расширение *.pl*, сокращение от «perl library». Мы по-прежнему используем это расширение для некоторых фрагментов кода Perl, загружаемых с помощью `do` *`FILE`* или `require`. Не будучи полноценным модулем, библиотечный файл обычно объявляет себя как отдельный пакет, чтобы связанные переменные и подпрограммы могли храниться вместе и не помешали случайно другим переменным в вашей программе. Обязательного расширения не существует; помимо *.pl* иногда встречаются другие, как разъясняется дальше в этой главе. Эти простые неструктурированные файлы в значительной мере заменены модулями.

module

Модуль Perl представляет собой библиотечный файл, удовлетворяющий неким особым соглашениям, которые позволяют одному или нескольким файлам, реализующим этот модуль, вовлекаться в действие с помощью одного объявления `use` во время компиляции. Имена файлов модулей всегда должны завершаться расширением *.pm*, поскольку это предполагается объявлением `use`. Объявление `use` также транслирует разделитель пакетов `::` в тот символ, который выполняет в данной системе функции разделителя каталогов, чтобы структура каталогов библиотеки Perl пользователя соответствовала структуре его пакета. Глава 11 описывает, как создавать собственные модули Perl.

class

Класс является просто модулем, в котором реализованы методы объектов, связанных с именем пакета модуля. Тем, кого интересуют объектно-ориентированные модули, следует обратиться к главе 12.

pragma

Прагма (*pragma*) является специальным модулем, который приводит в действие скрытые механизмы Perl. См. главу 31 «Модули прагм».

extension

Расширение (*extension*) – это модуль Perl, который помимо загрузки файла *.pm* загружает также библиотеку совместного доступа, реализующую семантику модуля на C или C++.

program

Программа (*program*) Perl представляет собой код, предназначенный для выполнения в качестве независимой сущности; ее называют также *сценарием* (*script*), когда хотят сказать, что не следует ожидать от нее слишком многого, *приложением* (*application*) – если она большая и сложная, *исполняемым модулем* (*executable*), когда вызывающему ее безразлично, на каком языке она написана, или *решением в масштабе предприятия* (*enterprise solution*) – если она стоит целого состояния. Программы Perl могут существовать в виде исходного кода, байт-кода или родного машинного кода. Если это то, что можно запустить из командной строки, мы называем это программой.

Обзор библиотеки Perl

Можно сэкономить огромное количество времени, предприняв попытку ознакомиться со стандартной библиотекой Perl, потому что совершенно незачем изобретать колесо заново. Предупреждаем, однако, что эта коллекция охватывает очень большой диапазон материала. Некоторые библиотеки могут быть крайне полезны, другие могут оказаться совершенно неподходящими для ваших нужд. Например, тому, кто пишет на 100-процентно чистом Perl, модули, поддерживающие динамическую загрузку расширений C и C++ не сильно помогут.

Perl рассчитывает найти библиотечные модули где-нибудь по маршруту поиска включаемых библиотек, `@INC`. Этот массив задает упорядоченный список каталогов, в которых Perl осуществляет поиск, когда загружается какой-нибудь библиотечный код с использованием ключевых слов `do`, `require` или `use`. Легко можно вывести список этих каталогов, вызвав Perl с ключом `-V` (от verbose) или таким простым кодом:

```
% perl -le "print foreach @INC"
/usr/libdata/perl5/sparc-openbsd/5.00503
/usr/local/libdata/perl5/sparc-openbsd/5.00503
/usr/libdata/perl5
/usr/local/libdata/perl5
/usr/local/libdata/perl5/site_perl/sparc-openbsd
/usr/libdata/perl5/site_perl/sparc-openbsd
/usr/local/libdata/perl5/site_perl
/usr/libdata/perl5/site_perl
.
```

Это лишь пример возможного вывода. В каждой установке Perl прописаны свои пути. Важно отметить, однако, что хотя содержание может отличаться

в зависимости от поставщика и политики инсталляции, которой придерживается сервер, можно быть уверенным, что все стандартные библиотеки будут установлены с Perl. Для того чтобы узнать, откуда файл был загружен фактически, посмотрите переменную `%INC`. Для файла модуля можно точно сказать, откуда Perl его берет, с помощью команды:

```
% perldoc -l MODULE
```

Если посмотреть на каталоги в `@INC` и их подкаталоги, то можно обнаружить несколько различных типов устанавливаемых файлов. У большинства имена оканчиваются на *.pm*, но у некоторых – на *.pl*,*.ph*,*.al* или *.so*. Наибольший интерес для вас должна представлять первая группа, поскольку суффикс *.pm* указывает на то, что файл является подлинным модулем Perl. Более подробно о них чуть ниже.

Несколько файлов, оканчивающихся на *.pl*, представляют собой старые библиотеки Perl, о которых мы говорили выше. Они включены для совместимости со старыми версиями Perl из 80-х и начала 90-х годов. Благодаря им код Perl, работавший, скажем, в 1990 году, должен по-прежнему нормально работать, не привлекая к себе внимания, даже если у вас установлена современная версия Perl. Создавая новый код, который использует стандартную библиотеку Perl, всегда следует по возможности предпочесть версию *.pm* любой версии *.pl*. Дело в том, что модули не так засоряют пространство имен, как это делают старые файлы *.pl*.

Одно замечание об использовании расширения *.pl*: оно означает библиотеку Perl, а не программу Perl. Хотя *.pl* иногда применяется для обозначения программ Perl на веб-серверах, чтобы различить выполняемые программы и статическое содержание в одном и том же каталоге, мы предлагаем вместо него употреблять для обозначения выполняемой программы Perl суффикс *.plx*. (Аналогичный совет относится к операционным системам, выбирающим интерпретаторы, основываясь на расширениях имен файлов.)

Файлы с расширениями *.al* являются небольшими фрагментами более крупных модулей и автоматически загружаются при обращении к их родительскому *.pm*-файлу. Если структура модуля строится с применением стандартного инструмента *h2xs*, поставляемого с Perl (и если не был указан флаг Perl `-A`), то процедура `make install` воспользуется модулем `AutoLoader`, чтобы создать эти файлы *.al* без вашего участия.

Файлы *.ph* были созданы стандартной программой *h2ph*, несколько устаревшим, но иногда все еще необходимым инструментом, который старается транслировать директивы препроцессора C в Perl. Получаемые файлы *.ph* содержат константы, иногда необходимые функциям низкого уровня, таким как `ioctl`, `fcntl` или `syscall`. (В настоящее время большинство этих значений может быть более удобным и переносимым способом получено в стандартных модулях, таких как `POSIX`, `Errno`, `Fcntl` или `Socket`.) См. в *perlinstall*, как установить эти не обязательные, но иногда важные компоненты.

И последнее расширение имени файла, с которым вы можете столкнуться, рассматривая эти каталоги, это *.so* (или другое, применяемое в системе для обозначения библиотек совместного доступа). Эти файлы *.so* являются зависимыми от платформы частями модулей расширений. Первоначально написанные на C или C++, эти модули были скомпилированы в динамически перемещаемый объектный код. Конечному пользователю, однако, не требуется знать об их существовании, поскольку они скрыты за интерфейсом модуля. Когда код пользователя говорит `require Module` или `use Module`, Perl загружает *Module.pm* и выполняет его, что позволяет модулю загрузить другие необходимые части, такие как *Module.so* или автоматически загружаемые компоненты *.al*. На самом деле модуль может загрузить все что угодно, – включив 582 других модуля. Он может загрузить весь CPAN, если ему того захочется, а может быть, и архивы *freshmeat.net* за последние пару лет.

Модуль в Perl – это не просто статический участок кода. Это активный агент, который определяет, как реализовать интерфейс в ваших интересах. Он может выполнять все стандартные соглашения, а может не выполнять. Он может делать любые вещи, с тем, чтобы исказить смысл оставшейся части вашей программы, вплоть до того, что оттранслирует ее в SPITBOL. Такого рода крючкотворство считается совершенно допустимым, если оно хорошо документировано. При использовании такого модуля Perl вы соглашаетесь на *его* контракт, а не на стандартный контракт Perl.

Поэтому старайтесь читать мелкий шрифт.

31

Модули прагм

Прагма (*pragma*) является особым типом модуля, воздействующим на фазу компиляции программы. Некоторые модули прагм (или просто прагмы) могут оказывать воздействие и на фазу выполнения программы. Рассматривайте их как советы компилятору. Поскольку они должны быть видимы на этапе компиляции, то и работают они только при вызове посредством `use` или `no`, потому что к тому времени, когда выполняется `require` или `do`, компиляция давно закончена.

Принято записывать имена прагм целиком строчными буквами, т. к. имена модулей на нижнем регистре зарезервированы для самого дистрибутива Perl. При написании собственных модулей используйте в имени модуля хотя бы одну заглавную букву, чтобы избежать конфликта с именами прагм.

В отличие от обычных модулей, действие прагм в большинстве случаев ограничено оставшейся частью самого внутреннего охватывающего блока, из которого они вызваны. Иными словами, они имеют лексическую область видимости, подобно переменным `my`. Обычно область лексической видимости внешнего блока охватывает любой вложенный в него блок, но внутренний блок может отменить прагму с лексической областью видимости из внешнего блока с помощью команды `no`:

```
use strict;
use integer;
{
    no strict 'refs';       # разрешить символические ссылки
    no integer;             # возобновить арифметику с плавающей точкой
    # ....
}
```

Более чем другие модули, поставляемые с Perl, прагмы образуют существенную и важную часть среды компиляции Perl. Трудно хорошо использовать компилятор, не зная, как передавать ему советы, поэтому мы потратим некоторые дополнительные усилия на то, чтобы описать прагмы.

Следует также знать, что мы часто применяем прагмы для прототипирования функций, которые затем кодируются в «реальном» синтаксисе. Поэтому в некоторых программах вы найдете осуждаемые прагмы, такие как `use attrs`, функциональность которых сейчас поддерживается непосредственно синтаксисом объявления подпрограммы. Аналогично `use vars` находится в процессе замены объявлениями `our`. А `use subs` может однажды быть заменена атрибутом `override` в обычном объявлении подпрограммы. Мы не очень спешим ломать старые способы работы, но полагаем, что новые способы красивее.

use attributes

```
sub afunc : method;
my $closure = sub : method { ... };

use attributes;
@attrlist = attributes::get(\&afunc);
```

Прагма `attributes` имеет два назначения. Первое заключается в том, чтобы обеспечить внутренний механизм объявления *списков атрибутов* (*attribute lists*), которые являются необязательными свойствами, связываемыми с объявлениями подпрограмм и (однажды будут) с объявлениями переменных. (Поскольку это внутренний механизм, обычно эта прагма не используется непосредственно.) Вторым назначением является предоставление возможности извлекать эти списки атрибутов на этапе исполнения с помощью вызова функции `attributes::get`. В этом качестве `attributes` представляет собой обычный модуль, а не прагму.

В настоящее время Perl обрабатывает лишь несколько встроенных атрибутов. Использование специфических для пакетов атрибутов обеспечивает экспериментальный механизм расширения, описанный в разделе «Package-specific Attribute Handling» страницы руководства *attributes*(3).

Установка атрибутов происходит на этапе компиляции; попытка установить неизвестный атрибут является ошибкой компиляции. (Ошибка может перехватываться `eval`, но все равно прекращает компиляцию в этом блоке `eval`.)

В настоящее время реализованы только три встроенных атрибута подпрограмм: `locked`, `method` и `lvalue`. Более подробное их обсуждение см. в главе 6 «Подпрограммы» и в главе 7 «Потоки». В данное время нет встроенных атрибутов переменных, подобных реализованным для подпрограмм, но мы подумываем о некоторых возможных, например `constant`.

Прагма `attributes` предоставляет две подпрограммы для общего пользования. Вы можете запросить их импорт.

get

Эта функция возвращает (возможно, пустой) список атрибутов при передаче одного входного параметра, являющегося ссылкой на подпрограмму или переменную. Эта функция создает исключительную ситуацию путем вызова `Carp::croak`, если ей переданы неверные аргументы.

reftype

Эта функция действует сходным со встроенной функцией `ref` образом, но всегда возвращает основной встроенный тип данных Perl для объекта ссылки независимо от того, что он может быть освящен в какой-то пакет.

Точные детали обработки атрибутов находятся в процессе изменения, поэтому лучше всего ознакомиться с электронной документацией, поставляемой с конкретной версией Perl.

use autouse

```
use autouse 'Carp' => qw(carp croak);
carp "this carp was predeclared and autoused";
```

Эта прагма предоставляет механизм для загрузки некоторого модуля на этапе исполнения по требованию, т. е. только тогда, когда вызывается некоторая функция из этого модуля. Это делается путем создания функции-заглушки, которая при вызове заменяет себя реальным вызовом. По духу это близко к тому, как работают стандартные модули `AutoLoader` и `SelfLoader`. Если сказать кратко, то это «хак» производительности, который помогает вашей программе Perl запускаться быстрее (в среднем) за счет отсутствия компиляции модулей, которые в данном прогоне могут вовсе не быть вызваны.

Действие `autouse` зависит от того, загружен ли модуль. Например, если модуль `Module` уже загружен, то объявление:

```
use autouse 'Module' => qw(func1 func2($;$) Module::func3);
```

эквивалентно простому импорту двух функций:

```
use Module qw(func1 func2);
```

При этом предполагается, что `Module` определяет `func2()` с прототипом `($;$)` и что у `func1()` и `func3()` нет прототипов. (Вообще говоря, при этом также предполагается, что `Module` использует стандартный метод `import` из `Exporter`; в противном случае возбуждается фатальная ошибка.) Во всяком случае полностью игнорируется `Module::func3`, поскольку она, предположительно, уже объявлена.

Если, с другой стороны, `Module` еще не загружен, когда анализируется прагма `autouse`, то прагма объявляет, что функции `func1` и `func2` находятся в текущем пакете. Она также объявляет функцию `Module::func3` (что можно рассматривать как умеренно антиобщественную деятельность, поскольку антиобщественные последствия отсутствия модуля `Module` еще сильнее). При вы-

зове эти функции обеспечивают загрузку модуля Module, а затем заменяют себя вызовами реальных функций, которые только что загружены.

Поскольку прагма autouse перемещает часть выполнения программы с этапа компиляции на этап исполнения, это может иметь неприятные последствия. Например, если загружаемый по требованию модуль имеет инициализирующий код, выполнение которого предполагается на раннем этапе, инициализация может произойти слишком поздно. Также ошибки в вашем коде могут оказаться не выявленными, если важные проверки перемещены с этапа компиляции на этап исполнения.

В частности, если прототип, указанный в строке autouse, неверен, это не станет известно до выполнения соответствующей функции (что может произойти через месяцы или годы, если функция вызывается редко). Для частичного смягчения этой проблемы можно во время разработки писать код так:

```
use Chase;
use autouse Chase => qw(hue($) cry(&$));
cry "this cry was predeclared and autoused";
```

Первая строка обеспечивает раннее обнаружение ошибок при задании аргументов. Когда программа перейдет из стадии разработки в стадию производства, можно закомментировать обычную загрузку модуля Chase и оставить только вызов автозагрузки. В результате будут достигнуты надежность при разработке и производительность при работе.

use base

```
use base qw(Mother Father);
```

Эта прагма позволяет программисту удобно определить производный класс на основе перечисленных родительских классов. Приведенное объявление примерно эквивалентно следующему коду:

```
BEGIN {
    require Mother;
    require Father;
    push @ISA, qw(Mother Father);
}
```

Прагма use base выполняет все необходимые вызовы require. Если в области видимости находится прагма strict 'vars', то use base позволяет (в конечном итоге) осуществить присвоение @ISA без необходимости вначале объявить our @ISA. (Поскольку прагма use base обрабатывается на этапе компиляции, лучше всего не пытаться изменять самому @ISA на этапе исполнения.)

Но помимо этого use base имеет еще одно свойство. Если какой-либо именованный базовый класс использует свойство полей, описываемое для прагмы use fields далее в этой главе, то тогда прагма инициализирует специальные атрибуты полей пакета из базового класса. (Множественное наследова-

ние классов полей не поддерживается. Прагма `use base` возбуждает исключительную ситуацию, если поля есть более чем в одном именованном базовом классе.)

Любой еще не загруженный базовый класс будет автоматически загружен посредством вызова `require`. Однако необходимость `require` пакета базового класса определяется не обычным изучением `%INC`, а отсутствием глобальной переменной `$VERSION` в базовом пакете. Этот прием не дает Perl повторно пытаться (с неудачным исходом) загружать базовый класс, который отсутствует в собственном загружаемом файле (из-за того, скажем, что он загружен как часть файла некоторого другого модуля). Если `$VERSION` не обнаружена после успешной загрузки файла, `use base` определит `$VERSION` в базовом пакете, присвоив ей значение «`-1, defined by base.pm`».

use blib

Из командной строки:

```
% perl -Mblib program [args...]
% perl -Mblib=DIR program [args...]
```

Из программы Perl:

```
use blib;
use blib 'DIR';
```

Эта прагма предназначена преимущественно для тестирования произвольных программ Perl с не инсталлированными версиями пакетов с помощью ключа командной строки Perl `-M`. При этом предполагается, что структура каталогов была создана с помощью стандартного модуля `ExtUtils::MakeMaker`.

Прагма ищет структуру каталогов *blib*, начиная с каталога с именем *DIR* (или текущего каталога, если другой не был задан), а если каталог *blib* там не найден, отрабатывает назад от каталога «`..`», просматривая до пяти уровней родительского каталога.

use bytes

```
use bytes;
no bytes;
```

Прагма `use bytes` отключает символьную семантику для оставшейся части лексической области видимости, в которой она находится. Для обращения действия `use bytes` в текущей лексической области видимости может применяться прагма `no bytes`.

Perl обычно предполагает присутствие символьной семантики в присутствии символьных данных (данных из источника, для которого отмечена определенная кодировка символов).

Чтобы понять различие между символьной и байтовой семантикой, обратитесь к главе 15 «Unicode». Небесполезной может оказаться поездка в Токио.

use charnames

```
use charnames HOW;
print "\N{CHARSPEC} is a funny character";
```

Эта прагма с лексической областью видимости включает интерполяцию именованных символов в строки с помощью обозначений `\N{CHARSPEC}`:

```
use charnames ':full';
print "\N{GREEK SMALL LETTER SIGMA} is called sigma.\n";

use charnames ':short';
print "\N{greek:Sigma} is an upper-case sigma.\n";

use charnames qw(cyrillic greek);
print "\N{sigma} is Greek sigma, and \N{be} is Cyrillic b.\n";
```

Прагма поддерживает аргументы *HOW* `:full` и `:short`, а также конкретные имена шрифтов (scripts).[1] Аргумент *HOW* определяет, как следует искать символ, заданный *CHARSPEC* в `\N{CHARSPEC}`}. Если присутствует `:full`, то вначале в таблицах символов Unicode отыскивается *CHARSPEC* как полное имя символа Unicode. Если присутствует `:short` и *CHARSPEC* имеет вид *SCRIPTNAME*: *CHARNAME*, то *CHARNAME* отыскивается как буква в *SCRIPTNAME*. Если *HOW* содержит конкретные имена скриптов, то разыскивается `CHARSPEC` в качестве отдельного *CHARNAME* в каждом из заданных шрифтов в указанном порядке.

Для поиска *CHARNAME* в заданном сценарии *SCRIPTNAME* прагма ищет в таблице стандартных имен Unicode шаблоны вида:

```
SCRIPTNAME CAPITAL LETTER CHARNAME
SCRIPTNAME SMALL LETTER CHARNAME
SCRIPTNAME LETTER CHARNAME
```

Если *CHARNAME* целиком находится в нижнем регистре (как в `N{sigma}`), то вариант `CAPITAL` игнорируется. В противном случае игнорируется вариант `SMALL`.

Вы можете написать собственный модуль, работающий, как прагма `charnames`, но по-иному определяющий имена символов. Однако предназначенный для этого интерфейс все еще является экспериментальным, поэтому для получения последних сведений обратитесь к странице руководства.

---

[1] Мы не имеем здесь в виду «скрипты» Perl. Здесь имеется в виду некоторый стиль записи букв, например латынь, греческий, кириллица. К несчастью, «script» является техническим термином, и мы не собираемся требовать от Unicode Consortium употребления другого термина.

use constant

```
use constant BUFFER_SIZE    => 4096;
use constant ONE_YEAR       => 365.2425 * 24 * 60 * 60;
use constant PI             => 4 * atan2 1, 1;
use constant DEBUGGING      => 0;
use constant ORACLE         => 'oracle@cs.indiana.edu';
use constant USERNAME       => scalar getpwuid($<);
use constant USERINFO       => getpwuid($<);

sub deg2rad { PI * $_[0] / 180 }

print "This line does nothing"      unless DEBUGGING;
# ссылки могут быть объявлены константами
use constant CHASH          => { foo => 42 };
use constant CARRAY         => [ 1,2,3,4 ];
use constant CPSEUDOHASH    => [ { foo => 1}, 42 ];
use constant CCODE          => sub { "bite $_[0]\n" };

print CHASH->{foo};
print CARRAY->[$i];
print CPSEUDOHASH->{foo};
print CCODE->("me");
print CHASH->[10];                          # ошибка этапа компиляции
```

Эта прагма объявляет именованный символ неизменной константой[1] с заданным скалярным или списковым значением. Для каждого символа должно быть отдельное объявление. Значения вычисляются в списковом контексте. Это можно отменить с помощью `scalar`, как мы делали выше.

Поскольку эти константы не предваряются символом $, их нельзя непосредственно интерполировать в строки, заключенные в двойные кавычки, но можно делать это косвенно:

```
print "The value of PI is @{[ PI ]}.\n";
```

Поскольку списковые константы возвращаются как списки, а не как массивы, индексирование константы со списковым значением нужно осуществлять с помощью дополнительных круглых скобок, как и любых списковых значений:

```
$homedir = USERINFO[7];             # НЕВЕРНО
$homedir = (USERINFO)[7];           # ok
```

Хотя имена констант рекомендуется составлять только из заглавных букв, чтобы выделить их и избежать конфликтов с другими ключевыми словами и именами подпрограмм, это всего лишь соглашение. Имена констант должны начинаться с буквы, но она не обязана быть заглавной.

---

[1] Реализованной как подпрограмма без аргументов, возвращающая каждый раз одну и ту же константу.

Константы не ограничиваются лексической областью видимости, в которой они встретились. Они представляют собой просто подпрограммы без аргументов в таблице символов пакета, в котором объявлены. Можно ссылаться на константу *CONST* из пакета `Other` как на `Other::` *CONST*. См. дополнительно о встраивании таких подпрограмм на этапе компиляции в разделе «Подставляемые функции-константы» главы 6.

Как и для всех прагм `use`, выполнение `use constant` происходит на этапе компиляции. Поэтому в лучшем случае окажется вводящим в заблуждение размещение объявления `constant` внутри условного оператора, например `if ($foo) { use constant ... }`.

Если опустить значение символа, то он будет давать значение `undef` в скалярном контексте и пустой список `()` в списковом. Но лучше объявить их непосредственно:

```
use constant CAMELIDS       => ();
use constant CAMEL_HOME     => undef;
```

Ограничения на use constant

Списковые константы в настоящее время не подставляются так, как скалярные константы. Невозможно также иметь подпрограмму или ключевое слово с таким же именем, как константа. И это, вероятно, Правильная Вещь.

Нельзя объявить одновременно больше одной константы:

```
use constant FOO => 4, BAR => 5;    # НЕВЕРНО
```

В результате будет объявлена константа `FOO`, которая возвращает список `(4, "BAR", 5)`. Вместо этого нужно записать:

```
use constant FOO => 4
use constant BAR => 5;
```

Неприятности могут возникнуть в случае применения констант в контексте, в котором автоматически заключаются в кавычки голые имена. (Это относится к любому вызову подпрограммы, не только к константам.) Например, нельзя сказать `$hash{`*CONSTANT*`}`, потому что *CONSTANT* будет интерпретироваться как строка. Используйте `$hash{`*CONSTANT* `()}` или `$hash{+`*CONSTANT*`}`, чтобы помешать действию механизма расстановки кавычек. Аналогично, поскольку оператор `=>` заключает в кавычки свой левый операнд, если он является голым именем, правильной будет запись *CONSTANT* `() => 'value'`, а не *CONSTANT* `=> 'value'`.

В дальнейшем можно использовать атрибут `constant` в объявлениях переменных:

```
my $PI : constant = 4 * atan2(1,1);
```

Такое объявление влечет все преимущества употребления переменной вместо подпрограммы. Оно имеет все недостатки, связанные с тем, что пока еще не реализовано.

use diagnostics

```
use diagnostics;            # включение для этапа компиляции
use diagnostics -verbose;

enable  diagnostics;        # включение для этапа исполнения
disable diagnostics;        # выключение для этапа исполнения
```

Эта прагма расширяет обычную сжатую диагностику и подавляет вывод дублирующихся предупредительных сообщений. Она дополняет краткие версии более объясняющими и привлекательными, которые можно видеть в главе 33 «Диагностические сообщения». Как и другие прагмы, она действует на этапе компиляции, а не только на этапе исполнения.

Если выполнить `use diagnostics` в начале программы, то будет автоматически включена опция командной строки Perl `-w` путем установки `$^W` в 1. Последующая компиляция будет осуществляться при усиленной диагностике. Сообщения будут по-прежнему выводиться на `STDERR`.

Нельзя применять `no diagnostics` для отключения сообщений на этапе компиляции из-за взаимодействия между задачами этапа исполнения и этапа компиляции и потому, что это вообще плохая идея. Однако на этапе исполнения можно управлять диагностикой с помощью методов `disable` и `enable`. (Не забудьте сначала выполнить `use`, иначе невозможно будет использовать эти методы.)

Флаг `-verbose` печатает введение к странице руководства *perldiag* перед выводом какой-либо диагностики. Можно установить переменную `$diagnostics::PRETTY` (раньше чем `use`), чтобы создать улучшенные escape-последовательности для программ постраничного вывода *less*(1) или *more*(1):

```
BEGIN { $diagnostics::PRETTY = 1 }
use diagnostics;
```

Предупреждения, отправляемые Perl и обнаруживаемые этой прагмой, выводятся только один раз. Это удобно, если вы находитесь в цикле, повторяющем одно и то же сообщение (например, неинициализированное значение). Предупреждения, создаваемые вручную, например при вызовах `warn` или `carp`, не попадают под действие этого механизма обнаружения дубликатов.

Вот несколько примеров использования прагмы `diagnostics`. Приводимый файл несомненно выдаст несколько ошибок как при компиляции, так и во время исполнения:

```
use diagnostics;
print NOWHERE "nothing\n";
print STDERR "\n\tThis message should be unadorned.\n";
```

```
warn "\tThis is a user warning";
print "\nDIAGNOSTIC TESTER: Please enter a <CR> here: ";
my $a, $b = scalar <STDIN>;
print "\n";
print $x/$y;
```

А вот вывод:

```
Parentheses missing around "my" list at diagtest line 6 (#1)

    (W parenthesis) You said something like

        my $foo, $bar = @_;

    when you meant

        my ($foo, $bar) = @_;

Remember that "my", "our", and "local" bind tighter than comma.

Name "main::NOWHERE" used only once: possible typo at diagtest line 2 (#2)

(W once) Typographical errors often show up as unique variable
    names.  If you had a good reason for having a unique name,
    then just mention it again somehow to suppress the message.
    The our declaration is provided for this purpose.

Name "main::b" used only once: possible typo at diagtest line 6 (#2)
Name "main::x" used only once: possible typo at diagtest line 8 (#2)
Name "main::y" used only once: possible typo at diagtest line 8 (#2)

Filehandle main::NOWHERE never opened at diagtest line 2 (#3)

(W unopened) An I/O operation was attempted on a filehandle that
    was never initialized.  You need to do an open() or a socket()
    call, or call a constructor from the FileHandle package.

        This message should be unadorned.
        This is a user warning at diagtest line 4.

DIAGNOSTIC TESTER: Please enter a <CR> here:
Use of uninitialized value in division (/) at diagtest line 8 (#4)

(W uninitialized) An undefined value was used as if it were
    already defined.  It was interpreted as a "" or a 0, but maybe
    it was a mistake.  To suppress this warning assign a defined
    value to your variables.

Illegal division by zero at diagtest line 8 (#5)

(F) You tried to divide a number by 0.  Either something was
    wrong in your logic, or you need to put a conditional in to
    guard against meaningless input.

Uncaught exception from user code:
        Illegal division by zero at diagtest line 8.
```

Диагностические сообщения происходят из файла *perldiag.pod*. Если обнаруживается обработчик `$SIG{__WARN__}`**, он будет действовать, но только после того как функция** `diagnostics::splainthis` **(перехватчик** `$SIG{__WARN__}`

в прагме) закончит свою работу с вашими предупреждениями. В настоящее время Perl не поддерживает стеки обработчиков, так что это лучшее, что мы сейчас можем сделать. Имеется переменная `$diagnostics::DEBUG`, которую можно установить, если вам уж очень любопытно, что же там такое перехватывается:

```
BEGIN { $diagnostics::DEBUG = 1 }
use diagnostics;
```

use fields

В модуле `Pet`:

```
package Pet;
use strict;
use fields qw(name weight _Pet_pid);
my $PID = 0;
sub new {
    my Pet $self = shift;
    unless (ref $self) {
        $self = fields::new($self);
        $self->{_Pet_pid} = "this is Pet's secret ID";
    }
    $self->{name} = "Hey, you!";
    $self->{weight} = 20;
    return $self;
}
1;
```

В отдельной программе *demopet*:

```
use Pet;
my Pet $rock = new Pet;           # типизированная лексическая переменная

$rock->{name}     = "quartz";
$rock->{weight}   = "2kg";
$rock->{_Pet_pid} = 1233;         # закрытый атрибут

$rock->{color}    = "blue";       # генерирует ошибку этапа компиляции
```

В модуле `Dog`:

```
package Dog;
use strict;
use base 'Pet';                   # наследовать поля и методы Pet
use fields qw(name pedigree);     # заменить атрибут Pet name,
                                  # добавить новый атрибут pedigree
use fields qw(wag _Dog_private);  # не общий с Pet
sub new {
    my $class = shift;
    my $self = fields::new($class);
    $self->SUPER::new();          # инициализировать базовые поля
```

```
    $self->{pedigree} = "none";   # инициализировать собственные поля
    return $self;
}
```

В отдельной программе *demodog*:

```
use Dog;

my Dog $spot = new Dog;            # типизованная лексическая переменная

$spot->{name}     = "Theloneus";   # не наследовался
$spot->{weight}   = "30lbs";       # наследовался
$spot->{pedigree} = "mutt";        # не наследовался

$spot->{color}    = "brown";       # генерирует ошибку этапа компиляции
$spot->{_Pet_pid} = 3324;          # генерирует ошибку этапа компиляции
```

Прагма `fields` дает метод, позволяющий присваивать значения полям класса, тип которых проверяется на этапе компиляции. Это ее свойство основывается на возможности, известной как псевдохеш: если типизированная лексическая переменная (`my Pet $rock`) содержит ссылку (объект `Pet`) и служит для доступа к элементу хеша (`$rock->{name}`) и если существует пакет с таким же именем, как объявленный тип, в котором поля класса организованы с использованием прагмы `fields`, то эта операция на этапе компиляции преобразуется в обращение к массиву, при условии, что указано допустимое поле.

Родственная прагма `base` объединяет поля базовых классов с полями, объявленными с помощью прагмы `fields`. Благодаря этому поля наследуются правильно.

Имена полей, начинающиеся с символа подчеркивания, становятся закрытыми в классе и невидимыми в его подклассах. Наследуемые поля могут переопределяться, но при этом генерируется предупреждение, если вывод предупреждений включен.

В результате всего этого мы получаем объекты с именованными полями, которые компактны, как массивы, и к которым столь же быстро осуществляется доступ. Однако это действует, только если обращаться к объектам через лексические переменные с надлежащими типами. Если переменные не типизированы, то доступ будет проверяться только на этапе исполнения, поэтому программа будет выполняться медленнее, т. к. ей придется обращаться и к хешу, и к массиву. Помимо объявлений полей поддерживаются следующие функции:

`new`

Функция `fields::new` создает псевдохеш и освящает его в указанный класс (который можно задать и передачей объекта этого класса). Объект создается с полями, объявленными ранее для этого класса с помощью прагмы `fields`. Это делает возможным написание таких конструкторов:

```
package Critter::Sounds;
use fields qw(cat dog bird);
```

```
    sub new {
        my Critter::Sounds $self = shift;
        $self = fields::new($self) unless ref $self;
        $self->{cat} = 'meow';                          # скалярный элемент
        @$self{'dog','bird'} = ('bark','tweet');        # срез
        return $self;
    }
```

phash

Функция fields::phash создает и инициализирует обычный (не освященный) псевдохеш. Всегда следует для создания псевдохешей применять эту функцию, а не делать это непосредственно, потому что реализация может измениться.

Если первый аргумент phash представляет собой ссылку на массив, то псевдохеш будет создан с ключами из этого массива. Если есть второй аргумент, то он тоже должен быть ссылкой на массив, элементы которого будут выступать в качестве значений. Если второй массив содержит меньше элементов, чем первый, то замыкающие элементы псевдохеша не будут инициализированы. Это особенно удобно для создания псевдохеша из аргументов подпрограммы:

```
sub dogtag {
    my $tag = fields::phash([qw(name rank ser_num)], [@_]);
}
```

Альтернативой является передача списка пар ключ/значение, которые будут использованы для создания псевдохеша:

```
my $tag = fields::phash(name => "Joe",
                        rank => "captain",
                        ser_num => 42);

my $pseudohash = fields::phash(%args);
```

Дополнительно о псевдохешах см. раздел «Псевдохеши» главы 8 «Ссылки».

В текущей реализации объявленные поля хранятся в хеше %FIELDS вызывающего пакета, но в будущих версиях это может измениться, поэтому для работы с полями лучше полагаться на интерфейс данной прагмы.

use filetest

```
$can_perhaps_read = -r "file";      # использовать биты режима
{
    use filetest 'access';          # стараться угадать
    $can_really_read = -r "file";
}
$can_perhaps_read = -r "file";      # использовать биты режима
```

Эта прагма с лексической областью видимости указывает компилятору на необходимость изменить режим работы унарных операторов тестирования файлов -r, -w, -x, -R, -W и -X, описанных в главе 3 «Унарные и бинарные операторы». В режиме по умолчанию эти операторы используют разряды режима, возвращаемые семейством вызовов stat. Однако не это бывает Правильным Делом, например, если файловая система поддерживает ACL (списки управления доступом). В таких средах, как AFS, где это имеет значение, прагма use filetest помогает операторам разрешений возвращать результаты, лучше согласующиеся с другими инструментами.

В случае применения этой прагмы может иметь место некоторое снижение скорости работы соответствующих операторов проверки файлов, потому что в некоторых системах расширенную функциональность приходится эмулировать.

Предупреждение: всякие мысли о применении проверок файлов для защиты данных совершенно безнадежны. Здесь открываются широкие возможности для возникновения «ситуации гонок», т. к. невозможно гарантировать, что права доступа не изменятся в промежутке между тестированием и реальным действием. Тот, кого хоть сколько-нибудь заботит безопасность, не станет использовать операторы тестирования файлов, чтобы решить, *будет* ли нечто работать. Вместо этого следует приступить к выполнению реальной операции, а затем проверить, успешно ли она выполнена (это нужно делать в любом случае). См. раздел «Обработка ошибок синхронизации» главы 23 «Защита данных».

use filetest 'access'

В настоящее время реализован импорт только access. При вызове use filetest 'access' включается использование *access*(2) или аналогичных системных вызовов при выполнении тестов файлов, а при вызове no filetest 'access' оно выключается. Эта расширенная функция тестирования применяется только тогда, когда операнд оператора (или, если хотите, аргумент унарной функции) действительно является именем файла, а не указателем файла.

use integer

```
use integer;
$x = 10/3;
# $x is now 3, not 3.33333333333333333
```

Эта прагма с лексической областью видимости указывает компилятору, что с этого места и до конца охватывающего блока он должен использовать операции с целыми числами. На многих машинах это не играет особой роли для большинства вычислений, но на тех редких архитектурах, где нет аппаратной поддержки действий над числами с плавающей точкой, производительность может резко измениться.

Обратите внимание, что эта прагма оказывает воздействие на некоторые числовые операции, но не на сами числа. Например, если выполнить такой код:

```
use integer;
$x = 1.8;
$y = $x + 1;
$z = -1.8;
```

то в результате мы получим: `$x == 1.8`, `$y == 2` и `$z == -1`. С `$z` это произошло потому, что унарный минус считается операцией, поэтому значение `1.8` усекается до `1`, перед тем как изменить знаковый разряд. Аналогично функции, предполагающие получение чисел с плавающей точкой, например `sqrt` или тригонометрические, продолжают принимать и возвращать действительные числа даже при действии `use integer`. Поэтому `sqrt(1.44)` равен `1.2`, но `0 + sqrt(1.44)` теперь просто `1`.

Используется родная арифметика целых чисел, предоставляемая вашим компилятором C. Это означает, что собственная семантика Perl для арифметических операций может быть не сохранена. Часто источником проблем является остаток при делении отрицательных чисел. Perl может делать это одним способом, а ваша машина – другим:

```
% perl -le 'print (4 % -3)'
-2
% perl -Minteger -le 'print (4 % -3)'
1
```

use less

```
use less;               # Все они НЕ РЕАЛИЗОВАНЫ!

use less 'CPU';
use less 'memory';
use less 'time';
use less 'disk';
use less 'fat';         # great with "use locale";
```

В настоящее время эта прагма не реализована, но когда-нибудь она будет давать компилятору, генератору кода или интерпретатору советы по поводу определенных компромиссов.

Не будет ошибкой попросить Perl использовать чего-либо меньше (less), даже если он пока не умеет этого делать.

use lib

```
use lib "$ENV{HOME}/libperl";   # добавить ~/libperl
no lib ".";                     # удалить cwd
```

Эта прагма упрощает работу с `@INC` во время компиляции. Обычно она применяется для добавления к маршруту поиска Perl новых каталогов, чтобы поз-

же команды do, require и use могли находить библиотечные файлы, которых нет в пути поиска Perl по умолчанию. Это особенно важно для use, потому что происходит и на этапе компиляции, а обычная установка @INC (т. е. на этапе исполнения) может произойти слишком поздно.

Параметры, передаваемые use lib, помещаются перед маршрутом поиска Perl. Сказать use lib *LIST* – это *почти* то же самое, что сказать BEGIN { unshift(@INC, *LIST*) }, но use lib *LIST* включает поддержку специфических для платформы каталогов. Для каждого каталога $dir из списка аргументов прагма lib проверяет также, существует ли каталог с именем *$dir/$archname/auto*. Если да, то каталог *$dir/$archname* полагается соответствующим специфическим для платформы каталогом и добавляется в @INC (перед $dir).

Во избежание избыточных добавлений, замедляющих обращение и расходующих память, дублирующиеся записи в конце @INC удаляются при добавлении.

Обычно нужно только *добавлять* каталоги в @INC. Если уж необходимо удалить каталоги из @INC, постарайтесь удалить только те, которые сами добавили, или те, в отношении которых вы уверены, что они не понадобятся другим модулям в вашей программе. Другие модули могли добавить каталоги к вашему @INC, которые нужны им для правильной работы.

Прагма no lib удаляет все вхождения указанных каталогов в @INC. Она также удаляет соответствующие специфические для платформы каталоги, как описано выше.

Загружаясь, прагма lib записывает текущее значение @INC в массив @lib::ORIG_INC, поэтому для восстановления исходного @INC скопируйте в него этот массив.

Хотя @INC обычно содержит точку (".."), текущий каталог – это не столь удобно, как может показаться. Во-первых, запись с точкой стоит в конце, а не в начале, чтобы модули, установленные в текущем каталоге, не отменяли системных версий. Если необходимо обратное, можно сказать use lib ".". Хуже, что это текущий каталог процесса Perl, а не каталог, в который был установлен сценарий, что делает его совершенно ненадежным. Если мы создаем программу и несколько модулей, которые она должна использовать, то все будет работать во время разработки и перестанет работать при запуске не из того каталога, где находятся файлы.

Один из выходов заключается в применении стандартного модуля FindBin:

```
use FindBin;                # куда был установлен сценарий?
use lib $FindBin::Bin;      # использовать этот каталог и для libs
```

Модуль FindBin пытается угадать полный путь к каталогу, в который была установлена программа текущего выполняющегося процесса. Не используйте его для защиты, потому что злоумышленные программы, если постараются, смогут его обмануть. Но если не ломать модуль специально, он будет работать так, как нужно. Модуль предоставляет переменную $FindBin::Bin (которую можно импортировать), содержащую предположение модуля о том, куда была установлена программа. Можете использовать прагму lib, чтобы

добавить этот каталог к своему @INC, создав путь, относящийся к исполняемому файлу.

Некоторые программы предполагают установку в каталог *bin* и ищут свои библиотечные модули в файлах, установленных в каталоге *lib*, на том же уровне, что и *bin*. Например, программы могут находиться в */usr/local/apache/bin* или */opt/perl/bin*, а библиотеки – в */usr/local/apache/lib* и */opt/perl/lib*. Следующий код ловко использует это обстоятельство:

```
use FindBin qw($Bin);
use lib "$Bin/../lib";
```

Обнаружив, что в нескольких разных программах задается одна и та же use lib, возможно, лучше установить переменную окружения PERL5LIB. См. описание переменной окружения PERL5LIB в главе 19 «Интерфейс командной строки».

```
# синтаксис для sh, bash, ksh или zsh
$ PERL5LIB=$HOME/perllib; export PERL5LIB

# синтаксис для csh или tcsh
% setenv PERL5LIB ~/perllib
```

Если необходимо работать с дополнительными каталогами только в этой программе, не меняя ее исходного кода, воспользуйтесь ключом -I:

```
% perl -I ~/perllib program-path args
```

Подробнее о применении -I в командной строке см. в главе 19.

use locale

```
@x = sort @y;        # сортировка в порядке ASCII
{
    use locale;
    @x = sort @y;    # использовать при сортировке национальные установки
}
@x = sort @y;        # снова сортировка в порядке ASCII
```

Эта прагма с лексической областью видимости говорит компилятору, что нужно включить (или выключить, если это прагма no locale) использование национальных установок из POSIX во встроенных операциях. При их включении функции Perl для преобразования регистра и механизм поиска по шаблону учитывают языковую среду, разрешая символы с диакритическими знаками и т. п. Если действует эта прагма, а ваша библиотека C знакома с национальными установками POSIX, то Perl учитывает установку LC_CTYPE в регулярных выражениях и установку LC_COLLATE при сравнении строк, например в sort.

Поскольку эти установки являются скорее формой национализма, а не интернационализма, то их использование может взаимодействовать с Unicode неожиданным образом. Подробнее о многоязычности (локализации) см. в главе 15.

use open

```
use open IN => ":crlf", OUT => ":raw";
```

Прагма open объявляет одну или несколько дисциплин ввода-вывода. Любой оператор open и readpipe (т. е. qx// или обратные штрихи), попадающий в область действия этой прагмы и не задавший собственной дисциплины, использует эти объявления по умолчанию. Эта прагма не оказывает влияния на open с явно заданным набором дисциплин и на sysopen в любом случае.

В настоящее время можно применять только две дисциплины, :raw и :crlf (хотя в скором времени ожидается подход дисциплины :utf8). В унаследованных системах, различающих два режима трансляции при открытии файлов, дисциплина :raw соответствует «двоичному режиму», а :crlf – «текстовому». (Эти две дисциплины в настоящее время представляют собой пустые операции на платформах, где пустой операцией является binmode, но только временно; см. функцию open в главе 29 «Функции» – там приведено более пространное описание семантики, которая предполагается нами у различных дисциплин.)

Полноценная поддержка дисциплин ввода-вывода в настоящее время не реализована. Когда они будут поддерживаться, эта прагма станет одним из интерфейсов объявления дисциплин по умолчанию для любого ввода-вывода. Тогда любые дисциплины по умолчанию, объявляемые этой прагмой, будут доступны как специальная дисциплина с именем «:DEFAULT» и использоваться в конструкторах указателей, поддерживающих задание дисциплин. Это позволит организовывать стек новых дисциплин над установленными по умолчанию.

```
open (FH, "<:para :DEFAULT", $file) or die "can't open $file: $!";
```

Когда будет завершена полная поддержка дисциплин ввода-вывода, все поддерживаемые дисциплины смогут работать на всех платформах.

use overload

В модуле Number:

```
package Number;
use overload "+" => \&myadd,
             "-" => \&mysub,
            "*=" => "multiply_by";
```

В пользовательской программе:

```
use Number;
$a = new Number 57;
$b = $a + 5;
```

Встроенные операторы хорошо работают со строками и числами, но имеют мало смысла в применении к ссылкам на объекты (поскольку, в отличие от C и C++, Perl не поддерживает арифметику указателей). Прагма `overload` позволяет переопределить значение этих встроенных операций для действий с объектами вашей собственной конструкции. В предыдущем примере вызов прагмы переопределяет три операции над объектами `Number`: сложение будет вызывать функцию `Number::myadd`, вычитание – функцию `Number::mysub`, а оператор мультипликативного присвоения будет вызывать метод `multiply_by` в классе `Number` (или одном из его базовых классов). Мы говорим об этих операторах, что они теперь *перегружены*, потому что на них возложен дополнительный смысл (а не потому, что у них теперь слишком много значений, хотя бывает и так).

Значительно подробнее о перегрузке сказано в главе 13 «Перегрузка».

use re

Эта прагма управляет применением регулярных выражений. Она может быть вызвана четырьмя способами: `'taint'` и `'eval'`, которые имеют лексическую область видимости, а также `'debug'` и `'debugcolor'`, которые ее не имеют.

```
use re 'taint';
# Содержимое $match меченое, если $dirty тоже меченая.
($match) = ($dirty =~ /^(.*)$/s);

# Разрешить интерполяцию кода:
use re 'eval';
$pat = '(?{ $var = 1 })';       # выполнение встроенного кода
/alpha${pat}omega/;             # даст ошибку при -T
                                # и мечености $pat

use re 'debug';                 # как "perl -Dr"
/^(.*)$/s;                      # выводить отладочную информацию
                                #     на этапах компиляции и исполнения

use re 'debugcolor';            # то же, что 'debug'
                                #    но с окрашенной выдачей
```

Когда действует `use re 'taint'` и объект регулярного выражения представляет собой меченую строку, нумерованные переменные регулярного выражения и значения, возвращаемые оператором `m//` в списковом контексте, являются помеченными. Это полезно, когда регулярные выражения над помеченными данными предназначены не для извлечения безопасных подстрок, а для осуществления других преобразований. См. обсуждение меченых данных в главе 23.

Если действует `use re 'eval'`, то в регулярном выражении разрешены утверждения, выполняющие код Perl и имеющие вид `(?{ ... })`, даже если регулярное выражение содержит интерполируемые переменные. Выполнение участков кода в результате интерполяции переменных в регулярное выражение обычно запрещается по причинам безопасности: не нужно, чтобы программы, которые читают шаблоны из файлов конфигурации, аргументы командной строки или поля форм CGI, вдруг начинали выполнять произвольный код, если только они не были специально разработаны для такой цели. Эта прагма разрешает интерполяцию только не меченых строк; меченые данные будут все равно вызывать создание исключительных ситуаций (если выполнение производится при включенной проверке меченых данных). См. также главу 5 «Поиск по шаблону» и главу 23.

Для этой прагмы интерполяция предварительно компилированных регулярных выражений (создаваемых оператором `qr//`) не считается интерполяцией переменных. Тем не менее, если мы строим шаблон `qr//`, требуется, чтобы действовала `use re 'eval'`, если в интерполированных строках содержатся кодовые утверждения. Например:

```
$code = '(?{ $n++ })';       # кодовое утверждение
$str  = '\b\w+\b' . $code;   # строить интерполируемую строку

$line =~ /$str/;     # здесь нужна use re 'eval'

$pat = qr/$str/;     # здесь тоже нужна use re 'eval'
$line =~ /$pat/;     # а здесь не нужна use re 'eval'
```

При действии `use re 'debug'` Perl выводит отладочные сообщения во время компиляции и во время выполнения регулярных выражений. Вывод получается такой же, как при выполнении «отладочного Perl» (скомпилированного с ключом `-DDEBUGGING` компилятора C) и выполнении программы Perl с ключом `-Dr` командной строки Perl. В зависимости от сложности шаблона результирующий вывод может быть огромным. Вызов `use re 'debugcolor'` осуществляет расцвечивание вывода, что может быть удобно, если терминал понимает последовательности, управляющие цветом. Установите в переменной окружения `PERL_RE_TC` список с разделителем-запятой соответствующих свойств *termcap*(5) для выделения. Дополнительные сведения см. в главе 20 «Отладчик».

use sigtrap

```
use sigtrap;
use sigtrap qw(stack-trace old-interface-signals);  # то же самое

use sigtrap qw(BUS SEGV PIPE ABRT);
use sigtrap qw(die INT QUIT);
use sigtrap qw(die normal-signals);
use sigtrap qw(die untrapped normal-signals);
use sigtrap qw(die untrapped normal-signals
               stack-trace any error-signals);
```

```
use sigtrap 'handler' => \&my_handler, 'normal-signals';
use sigtrap qw(handler my_handler normal-signals stack-trace error-signals);
```

Прагма `sigtrap` устанавливает некоторые простые обработчики сигналов, чтобы программисту не нужно было о них беспокоиться. Это полезно, если не перехваченный сигнал может вызвать нарушение работы программы, как бывает, если в программе есть блоки `END {}`, деструкторы объектов или другая обработка на выходе, которую нужно произвести независимо от способа завершения ее работы.

Прагма `sigtrap` предоставляет два простых обработчика сигналов. Один обеспечивает трассировку стека Perl, а другой создает обычную исключительную ситуацию посредством `die`. Можно также передать прагме для установки свой собственный обработчик. Можно заранее задать группы перехватываемых сигналов или передать свой собственный явный список сигналов. Дополнительно прагма может установить обработчики только для тех сигналов, которые не обработаны иным способом.

Аргументы, передаваемые в `use sigtrap`, обрабатываются по порядку. Когда встречается определенное пользователем имя сигнала или имя одного из предопределенных списков сигналов `sigtrap`, немедленно устанавливается обработчик. Опции воздействуют только на обработчики, устанавливаемые далее при обработке списка аргументов.

Обработчики сигналов

Эти опции определяют, какой обработчик будет использоваться для сигналов, устанавливаемых позже:

`stack-trace`

Этот обработчик, предоставляемый прагмой, выводит трассировку стека Perl на `STDERR`, а затем пытается выполнить дамп памяти.

`die`

Этот обработчик, предоставляемый прагмой, вызывает `die` через `Carp::croak` с сообщением, указывающим на перехваченный сигнал.

`handler` *`YOURHANDLER`*

`YOURHANDLER` будет играть роль обработчика для сигналов, устанавливаемых позже. *`YOURHANDLER`* может быть любым значением, допустимым для присвоения в `%SIG`. Помните, что внутри обработчика сигнала нельзя гарантировать правильную работу многих библиотечных вызовов C (особенно для стандартного ввода-вывода). Еще хуже, что трудно определить, какой код библиотеки C каким кодом Perl вызывается. (С другой стороны, многие из сигналов, которые перехватывает `sigtrap`, довольно зловредные – они все равно, в конце концов, с вами разделаются, так что не будет особого вреда в том, чтобы *попытаться* что-то сделать.)

Предопределенные списки сигналов

У прагмы sigtrap есть несколько встроенных списков сигналов, которые нужно перехватывать:

normal-signals

Обычные сигналы. Сигналы, с которыми обычно может столкнуться программа и которые по умолчанию вызывают ее завершение. Ими являются HUP, INT, PIPE и TERM.

error-signals

Сигналы ошибок. Это сигналы, которые обычно указывают на серьезные проблемы в интерпретаторе Perl или в программе пользователя. В их число входят ABRT, BUS, EMT, FPE, ILL, QUIT, SEGV, SYS и TRAP.

old-interface-signals

Сигналы старого интерфейса. Они по умолчанию перехватывались в старой версии интерфейса sigtrap. В их число входят ABRT, BUS, EMT, FPE, ILL, PIPE, QUIT, SEGV, SYS, TERM и TRAP. Если в use sigtrap не передаются сигналы или списки, то используется этот список.

Если на данной платформе не реализован некоторый сигнал, указанный в предопределенных списках, то имя этого сигнала будет молча проигнорировано. (Сам сигнал нельзя проигнорировать, поскольку он не существует.)

Прочие аргументы sigtrap

untrapped

Этот маркер подавляет установку обработчиков для сигналов, перечисляемых впоследствии, если они уже перехвачены или игнорируются.

any

Этот маркер устанавливает обработчики для всех сигналов, перечисляемых впоследствии. Это режим, устанавливаемый по умолчанию.

signal

Любой аргумент, который выглядит как имя сигнала (т. е. соответствует шаблону /^[A-Z][A-Z0-9]*$/), запрашивает обработку этого сигнала прагмой sigtrap.

number

Числовой аргумент требует, чтобы номер версии прагмы sigtrap был не меньше *number*. Это работает, как и в обычных модулях, в которых есть переменная пакета $VERSION:

```
% perl -Msigtrap -le 'print $sigtrap::VERSION'
1.02
```

Примеры sigtrap

Обеспечить трассировку стека для сигналов старого интерфейса:

```
use sigtrap;
```

То же самое, но более явно:

```
use sigtrap qw(stack-trace old-interface-signals);
```

Обеспечить трассировку стека только для четырех перечисленных сигналов:

```
use sigtrap qw(BUS SEGV PIPE ABRT);
```

Выполнить `die` при получении сигнала `INT` или `QUIT`:

```
use sigtrap qw(die INT QUIT);
```

Выполнить `die` при получении любого из сигналов: `HUP`, `INT`, `PIPE` или `TERM`:

```
use sigtrap qw(die normal-signals);
```

Выполнить `die` при получении любого из сигналов: `HUP`, `INT`, `PIPE` или `TERM` — но не изменяя режим для сигналов, которые уже перехвачены или игнорируются в каком-либо месте программы:

```
use sigtrap qw(die untrapped normal-signals);
```

Выполнить `die` при получении любого из не перехватываемых в данный момент сигналов из списка `normal-signals`; кроме того, обеспечить возврат в стеке Perl при получении любого из сигналов `error-signals`:

```
use sigtrap qw(die untrapped normal-signals
               stack-trace any error-signals);
```

Установить подпрограмму `my_handler` в качестве обработчика для `normal-signals`:

```
use sigtrap 'handler' => \&my_handler, 'normal-signals';
```

Установить `my_handler` в качестве обработчика для `normal-signals`; обеспечить возврат в стеке Perl при получении любого из сигналов `error-signals`:

```
use sigtrap qw(handler my_handler normal-signals
               stack-trace error-signals);
```

use strict

```
use strict;          # Установить все три структуры.

use strict "vars";   # Переменные должны быть предварительно объявлены.
use strict "refs";   # Нельзя использовать символические ссылки.
use strict "subs";   # Строки голых слов необходимо заключить в кавычки.
```

```
use strict;          # Установить все...
no strict "vars";    # ...затем отменить одну.
```

Эта прагма с лексической областью видимости изменяет некоторые основные правила, определяющие, что Perl считает допустимым кодом. Иногда эти ограничения кажутся слишком строгими для программирования простых задач, например, когда мы просто пытаемся «слепить» программу-фильтр в пять строк. Чем больше программа, тем больше строгости необходимо проявлять по отношению к ней.

В настоящее время строгости целесообразны по отношению к трем вещам: subs, vars и refs. Если список импорта не задан, предполагается установка всех трех ограничений.

strict 'refs'

Эта прагма генерирует ошибки этапа исполнения в случае применения символических ссылок, намеренного или случайного. Подробнее о них см. в главе 8.

```
use strict 'refs';

$ref = \$foo;        # Сохранить "реальную" (жесткую) ссылку.
print $$ref;         # Разыменование допускается.

$ref = "foo";        # Сохранение имени глобальной (пакетной) переменной.
print $$ref;         # НЕВЕРНО, ошибка времени исполнения при strict refs.
```

Символические ссылки подозрительны по ряду причин. Даже добропорядочные программисты могут на удивление легко допустить ошибочный их вызов; прагма strict 'refs' предохраняет от этого. В отличие от реальных ссылок, символические могут относиться только к глобальным переменным. Для них не ведется учет числа ссылок. Часто можно отыскать лучший способ сделать то же самое: вместо ссылки на символ в глобальной таблице символов используйте хеш как самостоятельную мини-таблицу символов. Это более эффективно, лучше читается и меньше подвержено ошибкам.

Тем не менее, некоторые виды допустимых действий в самом деле требуют прямого доступа к таблице глобальных символов пакета для имен переменных и функций. Например, может потребоваться исследовать список @EXPORT или надкласс @ISA некоторого пакета, имя которого заранее не известно. Либо может понадобиться установить массу вызовов функций, каждый из которых является псевдонимом одного и того же замыкания. Это как раз то, для чего символические ссылки хороши, но чтобы использовать их, когда действует use strict, необходимо предварительно снять ограничение "refs":

```
# создать группу методов доступа к атрибуту
for my $methname (qw/name rank serno/) {
    no strict 'refs';
    *$methname = sub { $_[0]->{ __PACKAGE__ . $methname };
}
```

strict 'vars'

При этом ограничении ошибка этапа исполнения возникает всякий раз при попытке доступа к переменной, которая не удовлетворяет хотя бы одному из следующих критериев:

- Предопределена в самом Perl, например, это `@ARGV`, `%ENV` и глобальные переменные пунктуации, такие как `$.` или `$_`.
- Объявлена через `our` (для глобальной) или `my` (для лексической).
- Импортирована из другого пакета. (Прагма `use vars` подделывает импорт, но можно обратиться к функции `our`.)
- Полностью квалифицирована с использованием имени пакета и удвоенного двоеточия – разделителя пакета.

Одного только оператора `local` недостаточно, чтобы удовлетворить `use strict 'vars'`, поскольку, несмотря на свое имя, этот оператор не изменяет глобальность переменной. Он просто дает переменной новое временное значение до конца блока на этапе исполнения. По-прежнему `our` необходима для объявления глобальной переменной и `my` – для объявления лексической переменной. Однако `our` можно локализовать:

```
local our $law = "martial";
```

На глобальные переменные, предопределенные в Perl, эти требования не распространяются. Это относится к глобальным для всей программы переменным (которые помещены в пакет `main`, подобно `@ARGV` или `$_`) и к переменным пакета, таким как `$a` и `$b`, обычно используемым функцией `sort`. Пакетные переменные, используемые модулями типа `Exporter`, все же нуждаются в объявлении с помощью `our`:

```
our @EXPORT_OK = qw(name rank serno);
```

strict 'subs'

Это ограничение заставляет Perl рассматривать все голые слова как синтаксические ошибки. *Голое слово* («bareword» или, в некоторых диалектах, «bearword») является любым «голым» именем или идентификатором, для которого нет иной интерпретации, вынуждаемой контекстом. (Контекст часто вынуждается соседним ключевым словом или лексемой либо предварительным объявлением рассматриваемого слова.) Исторически голые слова интерпретировались как строки без кавычек. Данное ограничение делает такую интерпретацию незаконной. Если вы намерены работать с именем как со строкой, заключите его в кавычки. Для того чтобы использовать его как вызов функции, необходимо предварительное объявление или применение круглых скобок.

В качестве особого случая принудительного контекста можно рассмотреть слово, появляющееся самостоятельно в фигурных скобках в левой части

оператора =>, которое рассматривается как заключенное в кавычки и потому не подлежащее данному ограничению.

```
use strict 'subs';

$x = whatever;      # НЕВЕРНО: ошибка "голого слова"!
$x = whatever();    # Однако это всегда действует.

sub whatever;       # Предварительное объявление функции.
$x = whatever;      # Теперь ok.

# Такое применение разрешено, поскольку => заключает в кавычки:
%hash = (red => 1, blue => 2, green => 3);

$rednum =  $hash{red};              # Ok, фигурные скобки заключают в кавычки.

# But not this one:
@coolnums = @hash{blue, green};     # НЕВЕРНО: ошибка "голого слова".
@coolnums = @hash{"blue", "green"}; # Ok, слова теперь в кавычках.
@coolnums = @hash{qw/blue green/};  # Аналогично.
```

use subs

```
use subs qw/winken blinken nod/;
@x = winken 3..10;
@x = nod blinken @x
```

Эта прагма предварительно объявляет все имена в списке аргументов как стандартные подпрограммы. Преимущество в том, что после этого можно использовать эти функции без круглых скобок как операторы списка, как если бы они были объявлены самим программистом. Это не всегда так же полезно, как полное объявление, поскольку делает невозможным использование прототипов или атрибутов, таких как:

```
sub winken(@);
sub blinken(\@) : locked;
sub nod($) : lvalue;
```

Основываясь на стандартном механизме импорта, прагма `use subs` имеет не лексическую область видимости, а область видимости пакета. Это значит, что объявления действуют во всем файле, в котором они появляются, но только в текущем пакете. Такие объявления нельзя отменить с помощью `no subs`.

use vars

```
use vars qw($frobbed @munge %seen);
```

Эта прагма, когда-то применявшаяся для объявления глобальных переменных, в настоящее время уступает модификатору `our`. Предшествующее объявление лучше реализуется с помощью:

```
our($frobbed, @munge, %seen);
```

или даже:

```
our $frobbed = "F";
our @munge = "A" .. $frobbed;
our %seen = ();
```

Вне зависимости от того, которое из них используется, помните, что все они относятся к глобальным переменным пакетов, а не к лексическим переменным файлов.

use warnings

```
use warnings;   # то же, что импорт "всех"
no warnings;    #  то же, что отмена импорта "всех"

use warnings::register;
if (warnings::enabled()) {
    warnings::warn("some warning");
}

if (warnings::enabled("void")) {
    warnings::warn("void", "some warning");
}
```

Эта прагма с лексической областью видимости допускает гибкое управление встроенными предупреждениями Perl, выдаваемыми как компилятором, так и системой времени исполнения.

Когда-то управлять обработкой предупреждений в программе Perl можно было только с помощью параметра командной строки `-w` или переменной `$^W`. Это полезно, но действует в духе «все или ничего». Опция `-w` приводит к тому, что включается вывод предупреждений в коде модулей, которые могли быть написаны кем-то другим, что может озадачивать вас и смущать первоначального автора. Применение `$^W` для включения или выключения сообщений в блоках кода может оказаться не очень удачным, поскольку действует только на этапе исполнения, а не во время компиляции.[1] Другая проблема состоит в том, что эта глобальная для всей программы переменная имеет динамическую, а не лексическую область видимости. Это означает, что если мы включаем ее в блоке, а затем вызываем из него другой код, то снова рискуем включить вывод предупреждений в коде, который разрабатывался без следования таким точным стандартам.

Прагма `warnings` обходит эти ограничения, поскольку является механизмом этапа компиляции с лексической видимостью, позволяющим осуществлять тонкий контроль над тем, где надо, а где не надо выводить предупреждения. Определена иерархия категорий предупредительных сообщений (рис. 31.1), обеспечивающая независимое друг от друга включение и выключение групп предупреждений. (Конкретное разделение на категории экспериментально

[1] Если, конечно, нет блоков `BEGIN`.

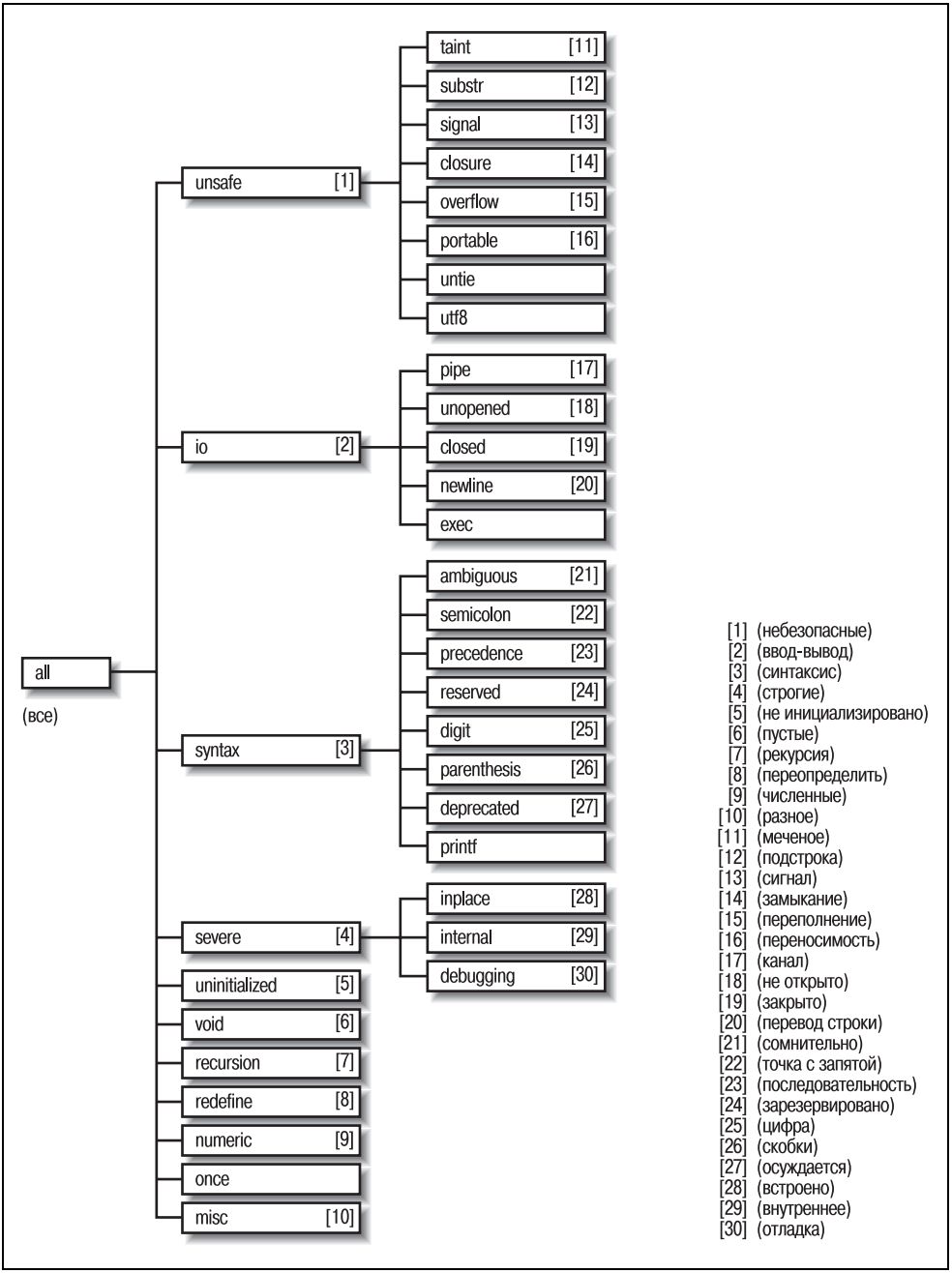


Рис. 31.1. Категории предупреждений Perl

и может измениться.) Эти категории можно объединять, передавая в use или no несколько аргументов:

```
use warnings qw(void redefine);
no  warnings qw(io syntax untie);
```

Если в данной области видимости действуют несколько экземпляров прагмы warnings, то их эффект складывается:

```
use warnings "void"; # Включена только "void".
...
use warnings "io";   # Включены "void" и "io".
...
no warnings "void";  # Включены только "io".
```

Чтобы превратить все предупреждения, включаемые некоторой прагмой warnings, в фатальные ошибки, поставьте слово FATAL перед списком импорта. Это удобно, если требуется, чтобы некоторое условие, обычно вызывающее предупреждение, прерывало выполнение программы. Допустим, например, что мы считаем обращение к неправильной строке как к числу (которое обычно становится в этом случае 0) совершенно недопустимым и хотим, чтобы это возмутительное событие убивало нашу программу. Попутно мы решаем, что использование неинициализированных значений там, где полагается быть действительным строкам или числам, тоже должно приводить программу к немедленному самоубийству:

```
{
    use warnings FATAL => qw(numeric uninitialized);
    $x = $y + $z;
}
```

Теперь, если $y или $z не инициализированы (т. е. содержат особое скалярное значение undef) или содержат строки, которые не очень хорошо преобразуются в числа, то вместо того, чтобы безмятежно двигаться дальше или, по крайности, слегка пожаловаться, если включена опция -w, наша программа создаст исключительную ситуацию. (Представьте себе Perl, работающий в режиме Python.) Если не обеспечить перехват исключительных ситуаций, то эта ошибка будет фатальной. Текст исключительной ситуации тот же, что появился бы в обычном предупреждении.

Прагма warnings игнорирует ключ командной строки -w и значение переменной $^W; установки прагмы имеют над ними приоритет. Однако флаг командной строки -W отменяет прагму, включая полный вывод предупреждений во всем коде программы, даже в том, который загружается с помощью do, require или use. Иными словами, с флагом -W Perl действует так, как если бы в каждом блоке программы присутствовала прагма use warnings 'all'. Можете представлять себе это как *lint*(1) для программ Perl. (Ознакомьтесь, однако, с электронной документацией по модулю B::Lint.) Флаг командной строки -X действует противоположным способом. Он делает вид, что в каждом блоке есть прагма no warnings 'all'.

Несколько функций предназначены в помощь авторам модулей, которые хотят, чтобы их функции в модулях вели себя как встроенные функции

в отношении лексической области видимости вызвавшего их модуля (другими словами, для того чтобы пользователи модуля имели возможность лексически включать и отключать предупреждения, которые может выводить модуль):

`warnings::register`

Регистрирует имя текущего модуля как новую категорию предупреждений, чтобы пользователи модуля могли выключать поступающие от него предупреждения.

`warnings::enabled(CATEGORY )`

Возвращает «истину», если категория предупреждений *CATEGORY* включена в лексической области видимости вызывающего модуля. В противном случае возвращает «ложь». Если *CATEGORY* не указана, используется имя текущего пакета.

`warnings::warn(CATEGORY, MESSAGE)`

Если в вызывающем модуле *CATEGORY* ***не*** установлена как `"FATAL"`, то выводит *MESSAGE* на `STDERR`. Если в вызывающем модуле *CATEGORY* установлена как `"FATAL"`, то выводит *MESSAGE* на `STDERR`, после чего умирает. Если *CATEGORY* не указана, используется имя текущего пакета.

32

Стандартные модули

В предыдущее издание этой книги было включено полное и окончательное описание всех модулей, входивших в стандартный дистрибутив Perl. Если бы мы сделали то же самое в данной книге, она обошлась бы вам вдвое дороже, не считая того, что пришлось бы расплачиваться с хиропрактиком за попытку дотащить книгу до дома. За последние несколько лет в стандартную поставку стало включаться все больше и больше модулей, и их насчитывается сейчас около двух сотен. Некоторые из них, например `CGI`, имеют весьма обширную собственную документацию. А если вы используете дистрибутив Perl ActiveState, то ваша стандартная библиотека еще более роскошна.

Поэтому мы теперь приводим полный список стандартных модулей, отсортированных по типам, вместе с кратким описанием того, что делает каждый из них. Затем мы выберем несколько наших любимых модулей и дадим примеры типичного их использования, сопровождаемые кратким описанием, позволяющим почувствовать, как с ними работать. Описания являются наводящими, а не исчерпывающими, и практически у всех модулей есть функции, здесь не описанные. Однако полная документация по всем стандартным модулям включается в каждый дистрибутив Perl, поэтому найти подробности для своей системы нетрудно с помощью программы *perldoc*, системной команды *man*(1) или своего любимого броузера. См. раздел «Электронная документация» в предисловии. Если найти документы не удается, обратитесь к Местному Знатоку, поскольку они (документы) почти наверняка установлены *где-то* в вашей системе. Даже если нет, всегда можно прочесть pod-документы прямо из самих модулей, поскольку вся документация по модулям встраивается в соответствующие файлы модулей (*.pm*), а pod разрабатывался так, чтобы его можно было читать. (В отличие, скажем, от HTML.)

Перечень по типу

Названия обычных модулей начинаются с заглавной буквы. Имена на нижнем регистре указывают на прагмы, которые описываются не в этой главе, а в главе 31 «Модули прагм».

Типы данных

Эти модули расширяют систему типов, имеющуюся (или отсутствующую) в Perl, различными способами.

Модуль	Описание
`Class::Struct`	Создание классов объектов Perl типа struct
`Constant`	Объявление константных скаляров и списков
`Integer`	Заставить Perl выполнять арифметику целых, а не с плавающей точкой
`Math::BigFloat`	Вычисления с плавающей точкой с числами произвольной длины
`Math::BigInt`	Вычисления с целыми числами произвольной длины
`Math::Complex`	Вычисления с комплексными числами и связанные с ними математические функции
`Math::Trig`	Загрузка многочисленных тригонометрических и связанных с ними функций
`Overload`	Перегрузка операторов Perl для объектов
`Symbol`	Работа с таблицами символов Perl и генерирование анонимных typeglob
`Time::Local`	Эффективное вычисление времени эпохи на основе `localtime` или `gmtime`

Обработка строк, обработка текста, синтаксический анализ и поиск

Эти модули обрабатывают текст.

Модуль	Описание
`Search::Dict`	Бинарный поиск строки в отсортированном тексте
`Text::Abbrev`	Создание таблицы аббревиатур из списка
`Text::ParseWords`	Разбор текста в список лексем или массив массивов
`Text::Soundex`	Использование алгоритма Кнута Soundex
`Text::Tabs`	Расширение символов табуляции и обратное действие аналогично *expand*(1) и *unexpand*(1)
`Text::Wrap`	Перенос строк для образования простых абзацев

Обработка опций, аргументов, параметров и файлов конфигурации

Эти модули обрабатывают командную строку.

Модуль	Описание
Getopt::Long	Обработка расширенных опций командной строки в длинном формате (--xxx)
Getopt::Std	Обработка односимвольных ключей при их слитном написании (-xyz)

Имена файлов, файловые системы и блокировка файлов

Эти модули предоставляют методы для доступа к файлам между различными платформами.

Модуль	Описание
Cwd	Получить путь к текущему рабочему каталогу
File::Basename	Переносимый разбор имени пути на каталог, основное имя и расширение
File::CheckTree	Запуск множества проверок файлов в группе деревьев каталогов
File::Compare	Переносимое сравнение содержимого файлов по именам или указателям
File::Copy	Переносимое копирование и перемещение файлов
File::DosGlob	Поиск файлов по шаблону имени в стиле Microsoft
File::Find	Обход деревьев файлов, как *find*(1)
File::Glob	Поиск файлов по шаблону имени в стиле Unix
File::Path	Переносимое создание и удаление группы каталогов
File::Spec	Использование переносимых операций с именами файлов (объектно-ориентированный интерфейс)
File::Spec::Functions	Использование переносимых операций с именами файлов (функциональный интерфейс)
File::Spec::Mac	Операции с именами файлов для файлов Mac OS
File::Spec::OS2	Операции с именами файлов для файлов OS/2
File::Spec::Unix	Операции с именами файлов для файлов Unix
File::Spec::VMS	Операции с именами файлов для файлов VMS
File::Spec::Win32	Операции с именами файлов для файлов Microsoft
File::stat	Замещение встроенных функций stat и lstat интерфейсом методов

Утилиты для работы с указателями файлов, указателями каталогов и потоковым вводом-выводом

Эти модули предоставляют объектно-ориентированный доступ к файлам, каталогам и межпроцессному взаимодействию.

Модуль	Описание
DirHandle	Использование объектных методов с указателями каталогов
FileCache	Держать одновременно открытыми больше файлов, чем позволяет операционная система
FileHandle	Применение объектных методов с указателями файлов
io	Предоставляет интерфейс для загрузки io::Dir, io::File, io::Handle, io::Pipe, io::Seekable и io::Socket
io::Dir	Использование методов объектов для указателей каталогов
io::File	Доступ к файловым методам объектов для указателей файлов
io::Handle	Применение общих методов объектов для указателей файлов
io::Pipe	Использование методов объектов для каналов
io::Poll	Объектный интерфейс к системному вызову *poll*(2)
io::Seekable	Доступ к объектным методам для объектов ввода-вывода с перемещаемой позицией
io::Select	Удобный ОО-интерфейс к системному вызову *select*(2)
SelectSaver	Запись и восстановление выбранного указателя файла

Многоязычность и национальные установки

Эти модули помогают работать с неамериканской кодировкой.

Модуль	Описание
Bytes	Использовать старую, ориентированную на байты семантику
Charnames	Определить имена символов для escape-последовательностей \N{name} в строковых литералах
I18N::Collate	Сравнивать 8-разрядные скалярные данные согласно текущим национальным установкам (не одобряется)
Locale	Использовать или игнорировать национальные установки POSIX для встроенных операций
utf8	Явно включить поддержку UTF-8 и Unicode

Интерфейсы с операционной системой

Эти модули позволяют «подправить» интерфейс с операционной системой.

Модуль	Описание
Fcntl	Загрузить определения библиотеки C *fcntl.h* как константы Perl
filetest	Управление операторами проверки файлов (-r, -w и т. п.) для нетрадиционных файловых систем
open	Установка дисциплин по умолчанию для вызовов функции open
POSIX	Использовать интерфейс Perl к POSIX 1003.1
Shell	Прозрачное выполнение команд оболочки в Perl
sigtrap	Включить простую обработку сигналов
Sys::Hostname	Переносимое использование всех возможных способов определения текущего имени хоста
Sys::Syslog	Работать с функциями библиотеки C *syslog*(3)
Time::gmtime	Заместить встроенную функцию gmtime интерфейсом метода
Time::localtime	Заместить встроенную функцию localtime интерфейсом метода
Time::tm	Создать внутренний объект, используемый Time::gmtime и Time::localtime
User::grent	Заместить встроенные функции getgr* интерфейсом метода
User::pwent	Заместить встроенные функции getpw* интерфейсом метода

Сетевое и межпроцессное взаимодействие

Эти модули предоставляют абстрактные интерфейсы вокруг базовых интерфейсов, о которых мы говорили в главе 16 «Межпроцессное взаимодействие».

Модуль	Описание
io::Socket	Обеспечивает общий объектный интерфейс к связи через сокеты
io::Socket::INET	Объектный интерфейс для сокетов домена Интернета
io::Socket::UNIX	Использовать объектный интерфейс для сокетов домена Unix (локальных)
IPC::Msg	Доступ к объектным методам для сообщений System V (SysV Msg IPC класс)
IPC::Open2	Открыть процесс для одновременного чтения и записи
IPC::Open3	Открыть процесс для чтения, записи и обработки ошибок
IPC::Semaphore	Использовать методы объектов для семафоров System V
IPC::SysV	Определить константы для всех механизмов IPC System V
Net::hostent	Заместить встроенные функции gethost* интерфейсом методов

Модуль	Описание
Net::netent	Заместить встроенные функции getnet* интерфейсом методов
Net::Ping	Проверить доступность удаленного хоста
Net::protoent	Заместить встроенные функции getproto* интерфейсом методов
Net::servent	Заместить встроенные функции getserv* интерфейсом методов
Socket	Загрузить определения и средства работы со структурами библиотеки C *socket.h*

World Wide Web

Эти модули предоставляют интерфейс к Сети. Возможно, вам доводилось слышать это название.

Модуль	Описание
CGI	Доступ к формам CGI и мощной автоматической генерации HTML
CGI::Apache	Позволить сценарию CGI работать под API Perl-Apache
CGI::Carp	Запись в журнал ошибок CGI *httpd*(8) (или другие)
CGI::Cookie	Устанавливать и получать cookie HTTP
CGI::Fast	Использовать протокол Fast CGI
CGI::Pretty	Создать красиво форматированный код HTML
CGI::Push	Создание CGI-сценариев push-технологий

Интерфейсы DBM

Эти модули загружают различные библиотеки для управления базами данных.

Модуль	Описание
AnyDBM_File	Обеспечить структуру для нескольких библиотек DBM
DB_File	Связанный доступ к библиотеке *db*(3) (версия 1.x Berkeley DB)
GDBM_File	Связанный доступ к библиотеке *gdbm*(3)
NDBM_File	Связанный доступ к библиотеке *ndbm*(3)
SDBM_File	Связанный доступ к файлам SDBM (простой DBM)

Пользовательские интерфейсы

Эти модули обеспечивают приличный OS CLI I/O API, ASAP.

Модуль	Описание
Term::Cap	Использовать библиотеку *termcap*(3)
Term::Complete	Программируемое завершение команд по спискам слов
Term::ReadLine	Работа с любым из нескольких пакетов readline

Аутентификация, безопасность и шифрование

Эти модули работают с «песочницами» (sandboxes).

Модуль	Описание
Opcode	Включить или отключить указанные коды операций при компиляции кода Perl для использования с модулем Safe
Ops	Ограничить небезопасные операции во время компиляции
Safe	Компилировать и выполнять код в ограниченных разделах

Язык Perl: расширения и свойства (extentions & internals)

Не путать с Намерениями и Расстройствами (Intentions & Externals).

Модуль	Описание
attributes	Установить или получить атрибуты подпрограммы или переменной
attrs	Установить или получить атрибуты переменной (устарел)
base	Установить наследование базового класса на этапе компиляции
Data::Dumper	Сериализация структур данных Perl
DB	Доступ к экспериментальному API отладчика Perl
Devel::DProf	Профилирование выполнения программы Perl
Devel::Peek	Загрузить средства отладки данных для программистов XS
diagnostics	Выводить словесные предупреждения и подавлять дубликаты
Dumpvalue	Обеспечить дамп данных на экран
English	Поддержка длинных имен для встроенных переменных пунктуации
Env	Доступ к переменным окружения в %ENV, как к обычным скалярам
Errno	Загрузить определения C-библиотеки *errno.h* и привязать переменную %!
Fatal	Заменить встроенные функции версиями, возбуждающими при неуспехе исключительные ситуации
fields	Объявление проверяемого на этапе компиляции доступа к атрибутам данных класса
less	Запросить у Perl меньшего расхода какого-либо ресурса (не реализована)
re	Изменить режим по умолчанию регулярных выражений
strict	Ограничить использование небрежных конструкций
subs	Предварительно объявлять имена подпрограмм в текущем пакете
vars	Предварительно объявлять глобальные переменные (устарел – см. our в главе 29 «Функции»

Классы с удобствами

Эти модули предоставляют базовые классы и другие удобства.

Модуль	Описание
`Tie::Array`	Предоставить базовый класс для связанных массивов
`Tie::Handle`	Предоставить определения базового класса для связанных указателей
`Tie::Hash`	Предоставить определения базового класса для связанных хешей
`Tie::RefHash`	Предоставить ссылки как ключи хеша
`Tie::Scalar`	Предоставить определения базового класса для связанных скаляров
`Tie::SubstrHash`	Хешировать таблицу фиксированного размера ключом фиксированной длины
`UNIVERSAL`	Предоставить базовый класс для *всех* классов (освященные ссылки)

Предупреждения и исключительные ситуации

Что делать, когда случаются неполадки.

Модуль	Описание
`Carp`	Предоставить программы, которые `warn` и `die` (предупреждают и погибают) с точки зрения вызывающего
`warnings`	Управление предупреждениями с лексической областью видимости

Поддержка документации

А документации, которую нужно поддерживать, очень много.

Модуль	Описание
`Pod::Checker`	Проверка документов pod на наличие синтаксических ошибок (используется *podchecker*(1))
`Pod::Functions`	Перечислить встроенные функции по типам
`Pod::Html`	Преобразовать файлы pod в HTML (используется *pod2html*(1))
`Pod::InputObjects`	Управление поддержкой документации
`Pod::Man`	Преобразование pod в формат *troff*(1) для системы *man*(1) (используется *pod2man*(1))
`Pod::Parser`	Предоставить базовый класс для создания фильтров и трансляторов pod
`Pod::Select`	Извлечь избранные разделы pod при вводе (используется *podselect*(1))
`Pod::Text`	Преобразовать данные pod в форматированный ASCII-текст (используется *pod2text*(1))

(*продолжение*)

Модуль	Описание
Pod::Text::Color	Преобразовать данные pod в форматированный текст ASCII с escape-последовательностями ANSI для цветов
Pod::Text::Termcap	Преобразовать данные pod в текст ASCII со специфическими для терминала escape-последовательностями
Pod::Usage	Вывести сообщение об использовании, полученное из встроенной документации pod

Поддержка установки модулей

Эти модули помогают другим модулям «прыгать через обручи».

Модуль	Описание
AutoLoader	Загрузка подпрограмм только по требованию
AutoSplit	Расщепить пакет для автозагрузки
autouse	Отложить загрузку модуля, пока не будет вызвана функция из него
blib	Использовать библиотеку из не установленной версии расширения MakeMaker
Config	Доступ к сведениям о конфигурации Perl
CPAN	Запрос, загрузка и компиляция модулей Perl с серверов CPAN
Devel::SelfStubber	Генерация заглушек для модуля посредством SelfLoader
DynaLoader	Динамически загружать библиотеки C и C++ как расширения Perl
Exporter	Реализовать метод импорта по умолчанию для обычных модулей
ExtUtils::Command	Предоставить средства для замены частых внешних команд в *Makefile*
ExtUtils::Embed	Обеспечить средства для встраивания Perl в программы C и C++
ExtUtils::Install	Установить файлы в системную библиотеку Perl
ExtUtils::Installed	Управление учетом установленных модулей
ExtUtils::Liblist	Определить, какие библиотеки использовать и как
ExtUtils::MakeMaker	Создать *Makefile* для расширения Perl
ExtUtils::Manifest	Загрузка средств записи и проверки файла *MANIFEST*
ExtUtils::Miniperl	Написать C-код для *perlmain.c*
ExtUtils::Mkbootstrap	Создать файл начальной загрузки для использования DynaLoader
ExtUtils::Mksymlists	Записать файлы опций компоновщика для динамического расширения

Модуль	Описание
ExtUtils::MM_Cygwin	Заместить методы режима Unix в ExtUtils::MakeMaker
ExtUtils::MM_OS2	Заместить методы режима Unix в ExtUtils::MakeMaker
ExtUtils::MM_Unix	Предоставить методы для использования ExtUtils::MakeMaker
ExtUtils::MM_VMS	Заместить методы режима Unix в ExtUtils::MakeMaker
ExtUtils::MM_Win32	Заместить методы режима Unix в ExtUtils::MakeMaker
ExtUtils::Packlist	Управление файлами *.packlist*
ExtUtils::testlib	Добавить каталоги *blib/** в @INC
FindBin	Найти каталог, в который была установлена выполняемая в данный момент программа Perl
lib	Работа с @INC на этапе компиляции
SelfLoader	Загрузка функций только по требованию
XSLoader	Динамическая загрузка библиотек C и C++ как расширений Perl

Поддержка разработки

Эти модули предназначены для учета времени и тестирования и позволяют увидеть, что код уже не выполняется более быстро и четко.

Модуль	Описание
Benchmark	Сравнить время выполнения альтернативных версий кода
Test	Использовать простую схему написания проверочных сценариев
Test::Harness	Запуск стандартных тестовых сценариев со статистикой

Компилятор и генератор кода Perl

Эти модули поддерживают различные серверы генерации кода для Perl.

Модуль	Описание
B	Загрузить генераторы кода Perl (a.k.a. «компиляторы Perl»)
B::Asmdata	Предоставить автогененрируемые данные об операциях Perl для производства байт-кода
B::Assembler	Ассемблировать байт-код Perl
B::Bblock	Обход базовых блоков в синтаксическом дереве
B::Bytecode	Применять сервер байт-кодов компилятора Perl
B::C	Использовать сервер трансляции C компилятора Perl
B::CC	Работать с оптимизированным сервером трансляции C компилятора Perl
B::Debug	Обход синтаксического дерева Perl с выводом отладочной информации об операциях

(*продолжение*)

Модуль	Описание
B::Deparse	Использовать сервер компилятора Perl для воспроизведения кода Perl
B::Disassembler	Дизассемблировать байт-код Perl
B::Lint	Выявление сомнительных конструкций
B::Showlex	Показать лексические переменные, используемые в функциях и файлах
B::Stash	Показать, какие сокровища загружены
B::Terse	Обход синтаксического дерева Perl с выводом краткой информации об операциях
B::Xref	Генерация отчетов о перекрестных ссылках для программ Perl
ByteLoader	Загрузка компилированного в байты кода Perl
O	Предоставление общего интерфейса к серверам компилятора Perl

Модули для систем Microsoft

Обладатели дистрибутива Perl для систем Microsoft от ActiveState имеют доступ к следующим модулям, включаемым только для Microsoft. Если вы просто добыли стандартный дистрибутив исходного кода (возможно, собираясь компилировать его в среде эмуляции Unix Cygwin), но работаете на платформе Intel под Windows, то можете получить все эти модули из CPAN.

Модуль	Описание
Win32::ChangeNotify	Отслеживание событий, связанных с файлами и каталогами
Win32::Console	Использовать функции Win32 для консоли и символьного режима
Win32::Event	Доступ к объектам событий Win32 из Perl
Win32::EventLog	Обработка журналов событий Win32 из Perl
Win32::File	Управление атрибутами файлов в Perl
Win32::FileSecurity	Управление списками контроля доступа FileSecurity в Perl
Win32::IPC	Загрузка базового класса для объектов синхронизации Win32
Win32::Internet	Доступ к функциям *WININET.DLL*
Win32::Mutex	Работать с объектами мьютексов Win32 из Perl
Win32::NetAdmin	Управление сетевыми группами и пользователями в Perl
Win32::NetResource	Управление сетевыми ресурсами в Perl
Win32::ODBC	Применять расширение ODBC в Win32
Win32::OLE	Доступ к расширениям автоматизации OLE

Модуль	Описание
Win32::OLE::Const	Извлечь определения констант из *TypeLib*
Win32::OLE::Enum	Использовать объекты Collection автоматизации OLE
Win32::OLE::NLS	Включить поддержку национальных языков OLE
Win32::OLE::Variant	Создание и модификация переменных OLE Variant
Win32::PerfLib	Доступ к счетчикам системного монитора Windows NT
Win32::Process	Создание процессов и управление ими
Win32::Semaphore	Использовать объекты семафоров Win32
Win32::Service	Управление системными сервисами
Win32::Sound	Работа со звуками Windows
Win32::TieRegistry	Работа с реестром (а что вы там забыли?)
Win32API::File	Доступ к низкоуровневым вызовам системного API Win32 для файлов и каталогов
Win32API::Net	Управление учетными записями сетевых клиентов Windows NT
Win32API::Registry	Доступ к низкоуровневым вызовам системного API Win32 из *WINREG.H*

Benchmark

```
use Benchmark qw(timethese cmpthese timeit countit timestr);

# Код можно передать в виде строк:
timethese $count, {
    'Name1' => '...code1...',
    'Name2' => '...code2...',
};

# Или в виде ссылок на подпрограммы:
timethese $count, {
    'Name1' => sub { ...code1... },
    'Name2' => sub { ...code2... },
};

cmpthese $count, {
    'Name1' => '...code1...',
    'Name2' => '...code2...',
};

$t = timeit $count, '...code...';
print "$count loops of code took:", timestr($t), "\n";

$t = countit $time, '...code...';
$count = $t->iters;
print "$count loops of code took:", timestr($t), "\n";
```

Модуль Benchmark помогает определить, какой из нескольких возможных вариантов кода выполняется быстрее всего. Функция timethese прогоняет заданные участки кода требуемое число раз и сообщает, сколько времени заняло выполнение каждого участка. Если вызвать таким же способом cmpthese, то можно получить прекрасно отсортированную диаграмму.

Участки кода могут (и даже должны, если программист использует лексические переменные из области видимости вызова) быть заданы как ссылки на функции, а не как строки, но расходы, связанные с вызовом, могут повлиять на результаты измерений. Если запрашивается число итераций, недостаточное для получения точных данных, функция выдает предупреждение.

Имеются интерфейсы более низкого уровня, с помощью которых можно запустить один участок кода для заданного числа итераций (timeit) или на заданное число секунд (countit). Эти функции возвращают объекты Benchmark (см. описание в электронной документации). Если параметр countit задан, то мы знаем, что она будет работать как раз столько времени, чтобы избежать предупреждений, поскольку задано минимальное время прогона.

Чтобы использовать модуль Benchmark с наибольшей эффективностью, нужно немало попрактиковаться. Обычно недостаточно выполнить пару различных алгоритмов с одним и тем же набором данных, поскольку результат будет отражать только то, как эти алгоритмы справились с данным конкретным набором данных. Чтобы получить лучшее представление для общего случая, необходимо прогнать несколько наборов тестов с различными данными.

Допустим, например, требуется узнать, как лучше всего скопировать строку без двух последних символов. Мы решаем, что это можно сделать четырьмя способами (хотя, конечно, есть и другие): дважды выполнить chop, копировать и сделать замену или использовать substr в левой или правой части присваивания. Проверяем эти алгоритмы на строках длиной 2, 200 и 20_000:

```
use Benchmark qw/countit cmpthese/;
sub run($) { countit(5, @_) }
for $size (2, 200, 20_000) {
    $s = "." x $len;
    print "\nDATASIZE = $size\n";
    cmpthese {
        chop2   => run q{
            $t = $s; chop $t; chop $t;
        },
        subs    => run q{
            ($t = $s) =~ s/..\Z//s;
        },
        lsubstr => run q{
             $t = $s; substr($t, -2) = '';
        },
        rsubstr => run q{
             $t = substr($s, 0, length($s)-2);
        },
    };
}
```

Результат приведен ниже:

```
DATASIZE = 2
            Rate    subs lsubstr   chop2 rsubstr
subs    181399/s      - -    -15%    -46%    -53%
lsubstr 214655/s     18%      - -    -37%    -44%
chop2   338477/s     87%     58%      - -    -12%
rsubstr 384487/s    112%     79%     14%      - -

DATASIZE = 200
            Rate    subs lsubstr rsubstr   chop2
subs    200967/s      - -    -18%    -24%    -34%
lsubstr 246468/s     23%      - -     -7%    -19%
rsubstr 264428/s     32%      7%      - -    -13%
chop2   304818/s     52%     24%     15%      - -

DATASIZE = 20000
          Rate rsubstr    subs lsubstr   chop2
rsubstr 5271/s      - -    -42%    -43%    -45%
subs    9087/s     72%      - -     -2%     -6%
lsubstr 9260/s     76%      2%      - -     -4%
chop2   9660/s     83%      6%      4%      - -
```

На маленьких наборах данных алгоритм "rsubstr" работает на 14% быстрее, чем алгоритм "chop2", а на больших наборах данных он работает на 45% медленнее. На пустых наборах данных (здесь не показанных) самым быстрым оказывается механизм подстановки. Поэтому часто лучшего решения на все случаи жизни нет, и даже эти замеры времени говорят не обо всем, т. к. еще они зависят от операционной системы и библиотеки C, с которой компилировался Perl. Что хорошо в одном случае, в другом может оказаться плохо. Требуется время, чтобы приобрести достаточное мастерство в измерении производительности. Пока этого не произойдет, выводы могут оказаться ложными.

Carp

```
use Carp;
croak "We're outta here!";

use Carp qw(:DEFAULT cluck);
cluck "This is how we got here!";
```

Модуль Carp позволяет создавать модули, функции в которых сообщают об ошибках так же, как это делают встроенные операторы – с точки зрения пользователей модуля. Модуль Carp предоставляет программы, которые используются сходным со стандартными встроенными функциями warn и die образом, но изменяют имя файла и номер строки, поэтому выглядят как ошибки, произошедшие в коде пользователя, а не в вашем. Короче, Carp прекрасно помогает свалить вину на другого.

В модуле есть четыре функции. Функция carp работает, как оператор warn, но с данными об имени файла и номере строки, относящимися к вызвавшему (модулю). Функция croak работает, как die, возбуждая исключительную ситуацию, но, опять-таки, предоставляя данные, относящиеся к вызвавшему. Те, кто предпочитает длинные жалобы, могут применять cluck и confess вместо carp и croak соответственно, получая полный отчет по стеку о том, кто кого вызывал и с какими аргументами. Следует явно импортировать cluck, поскольку в обычных условиях она не экспортируется. Обычно люди почему-то не хотят видеть стек целиком в случае простых предупреждений.

CGI

```
use CGI qw(:standard);
$who   = param("Name");
$phone = param("Number");
@picks = param("Choices");
```

Модуль CGI помогает работать с формами HTML, особенно многозвенными, когда важно передавать состояние от одного шага к другому. Приведенный выше крайне простой пример предполагает обработку формы с двумя параметрами, принимающими одиночные значения, например, текстовыми полями или переключателями (radio buttons), и одним, принимающим несколько значений, например, прокручиваемым списком с атрибутом "MULTIPLE". Данный модуль на несколько порядков сложнее, чем наш пример, и поддерживает удобную обработку файлов cookies, сохраняемые значения при переходе по страницам корзин покупок и динамическую генерацию списков и таблиц HTML, которые можно получать из баз данных, а также многое другое. Обеспечивается также ускоренное выполнение прекомпилированных сценариев Perl с помощью средства Apache mod_perl. Обо всем этом можно прочесть в изданной O'Reilly книге «Writing Apache Modules with Perl and C» Линкольна Стайна (Lincoln Stein) и Дуга Макичерна (Doug MacEachern).

CGI::Carp

```
use CGI::Carp;
warn "This is a complaint";     # Проштамповать именем программы и датой.
die "But this one is serious";  # Но не вызывать ошибок Server 500.

use CGI::Carp qw(carpout);      # Импортировать эту функцию.
open(LOG, ">>/var/tmp/mycgi-log")
    or die "Can't append to mycgi-log: $!\n";
carpout(*LOG);                  # Теперь использует отдельный журнал ошибок.

use CGI::Carp qw(fatalsToBrowser);
die "Сообщения о фатальных ошибках теперь посылаются также на броузер";
```

Модуль CGI::Carp предоставляет версии встроенных функций Perl warn и die Perl, а также функции carp, cluck, confess и croak модуля Carp, которые более

информативны и надежны. Они более информативны, поскольку каждое сообщение содержит дату, время и название программы, выдавшей сообщение, что удобно при совместном использовании журнального файла сотнями различных программ, которые одновременно пишут в него тысячи сообщений.

Этот модуль также добрее к путешествующим по Интернету, т. к. преждевременная смерть в сценарии CGI влечет непонятную ошибку "Server 500", когда правильный заголовок HTTP попадает на сервер уже после того, как сама программа «помрет», а данный модуль гарантирует, что этого не случится. Функция carpout переадресует все предупреждения и ошибки по заданному указателю файла. Директива fatalsToBrowser отправляет копию таких сообщений и на броузер пользователя. Эти средства облегчают отладку сценариев CGI.

Class::Struct

```
use Class::Struct;

struct Manager => {          # Создает конструктор Manager->new().
    name    => '$',          # Теперь метод name() обращается к скалярной величине.
    salary  => '$',          # То же делает salary().
    started => '$',          # То же делает started().
};

struct Shoppe => {           # Создает конструктор Shoppe->new().
    owner   => '$',          # Теперь метод owner()обращается к скаляру.
    addrs   => '@',          # а метод addrs() обращается к массиву.
    stock   => '%',          # а метод stock() обращается к хешу.
    boss    => 'Manager',    # Инициализирует с помощью Manager->new().
};

$store = Shoppe->new();
$store->owner('Abdul Alhazred');
$store->addrs(0, 'Miskatonic University');
$store->addrs(1, 'Innsmouth, Mass.');
$store->stock("books", 208);
$store->stock("charms", 3);
$store->stock("potions", "none");
$store->boss->name('Prof L. P. Haitch');
$store->boss->salary('madness');
$store->boss->started(scalar localtime);
```

Модуль Class::Struct дает возможность «объявить» класс, в объектах которого поля имеют заданные типы. Функция, которая осуществляет это, называется struct. Поскольку структуры и записи не являются базовыми типами Perl, то всякий раз, когда необходимо создать класс для объектов данных, имеющих вид записей, приходится определять конструктор и методы доступа для каждого поля данных, которые иногда называют «оборачивающими» методами (wrappers). Функция struct модуля Class::Struct отгоняет эту

тоску, создавая класс «на лету». Нужно только сообщить ей, какие данные-члены должны существовать, и указать их типы. Функция создает в пакете, задаваемом первым аргументом, конструктор с именем new, а также методы доступа для каждого члена, указанные во втором аргументе, который должен быть ссылкой на хеш.

Типы полей задаются как встроенные с использованием обычных символов "$", "@", "%" и "&" либо как другой класс с помощью имени класса. Тип каждого поля будет соблюдаться при попытке присвоить значение.

Многие стандартные модули используют Class::Struct для создания объектов и методов доступа, в том числе Net::hostent и User::pwent, исходный код которых можно изучить в качестве примера. Кроме того, в модулях CPAN Tie::SecureHash и Class::Multimethods реализован более развитый подход к автоматической генерации классов и методов доступа. См. раздел «Управление данными экземпляров» главы 12 «Объекты».

Config

```
use Config;
if ($Config{cc} =~ /gcc/) {
    print "This perl was built by GNU C.\n";
}
use Config qw(myconfig config_sh config_vars);
print myconfig();    # как perl -V без шаблона
print config_sh();   # выдает абсолютно все

config_vars qw/osname osvers archname/;
```

Конфигурирующий механизм, который компилирует и устанавливает Perl, собирает массу информации о вашей системе. Модуль Config по умолчанию экспортирует связанный хеш с именем %Config, предоставляющий доступ к более чем 900 различным параметрам конфигурации. (Эти значения доступны также через ключ командной строки Perl -V: *PATTERN*.) Config предоставляет также три функции, которые дают доступ к этим значениям, более напоминающий оболочку, как показано выше. Например, последний вызов мог вывести следующее:

```
osname='openbsd';
osvers='2.6';
archname='OpenBSD.sparc-openbsd';
```

В электронной документации по модулю описываются переменные конфигурации и их возможные значения. Учтите, что если переместить исполняемый файл *perl* в систему, отличную от той, в которой он компилировался, то эти значения могут не отражать реальности; например, если выполнять двоичный файл, сделанный для Linux или Solaris, на системе BSD.

CPAN

```
# Получить интерактивную оболочку CPAN.
% perl -MCPAN -e shell

# Попросить рекомендации по апгрейду.
% perl -MCPAN -e 'CPAN::Shell->r'

# Установить заданный модуль в пакетном режиме.
% perl -MCPAN -e "install Class::Multimethods"
```

Модуль CPAN представляет собой автоматизированный, дружественный для пользователя интерфейс к архиву Perl CPAN, описанному в главе 22. В отличие от большинства других модулей, его следует вызывать из командной строки как маленькую программу. При первом обращении к нему модуль просит ввести адрес зеркального сервера CPAN, используемого по умолчанию, и другие требующиеся ему данные. После этого можно запустить его интерактивную оболочку, чтобы формировать запросы и выбирать модули для установки, спрашивать у модуля рекомендации по поводу модулей, требующих обновления, или просто заставить его установить конкретный модуль.

Cwd

```
use Cwd;
$dir = getcwd();     # Иде я?

use Cwd 'chdir';
chdir "/tmp";        # Обновляет $ENV{PWD}.

use Cwd 'realpath';
print realpath("/usr////spool//mqueue/../");  # выводит /var/spool
```

Модуль Cwd предоставляет не зависимые от платформы функции для определения текущего рабочего каталога процесса. Это лучше, чем вызов через оболочку *pwd*(1), поскольку в системах, не удовлетворяющих POSIX, такой команды может не быть, а Perl работает не только на платформах POSIX. Функция getcwd, которая экспортируется по умолчанию, возвращает текущий рабочий каталог с помощью того механизма, который считается самым надежным для данной платформы. Если импортируется функция chdir, то она замещает встроенный оператор и поддерживает переменную окружения $ENV{PWD}; выполняемые в дальнейшем команды, использующие эту переменную, будут единообразно «смотреть на окружающий мир». Функция realpath разрешает свой аргумент, который может содержать символические ссылки и относительные пути, возвращая полный путь каталога в канонической форме, как *realpath*(3).

Data::Dumper

```
use Data::Dumper;
print Dumper($store);
```

В случае применения с примером из Class::Struct выводится следующее:

```
$VAR1 = bless( {
             'Shoppe::owner' => 'Abdul Alhazred',
             'Shoppe::stock' => {
                                  'charms' => 3,
                                  'books' => 208,
                                  'potions' => 'none'
                                },
             'Shoppe::boss' => bless( {
                                        'Manager::name' =>
                                            'Prof L. P. Haitch',
                                        'Manager::salary' =>
                                            'madness',
                                        'Manager::started' =>
                                            'Sat Apr  1 16:18:13 2000'
                                      }, 'Manager' ),
             'Shoppe::addrs' => [
                                  'Miskatonic University',
                                  'Innsmouth, Mass.'
                                ]
           }, 'Shoppe' );
```

Функция Dumper модуля Data::Dumper принимает список скаляров (в том числе ссылки, которые могут указывать на объекты) и возвращает строку, которую можно вывести или выполнить через eval, являющуюся точной и верной копией оригинала. С ее помощью можно сохранять структуру данных в дисковом файле или файле DBM либо передать другому процессу через канал или сокет. Этот модуль можно использовать с MLDBM из CPAN и DB_File для реализации файла DBM, который может прозрачно сохранять сложные значения данных, а не просто обычные строки. Другими модулями для преобразования в строки (или *сериализации* (*serialization*), или *пересылки* (*marshalling*)) являются Storable и FreezeThaw, которые находятся в CPAN.

DB_File

```
use DB_File;
```

Связать хеш с файлом DBM:

```
tie(%hash, "DB_File", $filename)     # Открыть базу данных.
    or die "Can't open $filename: $!";

$v = $hash{"key"};                   # Извлечь из базы данных.
```

```
$hash{"key"} = "value";              # Поместить в базу данных.
untie %hash;
```

Привязать хеш к файлу двоичного дерева, но обращаться как к обычному хешу DBM:

```
tie(%hash, "DB_File", "mytree", O_RDWR|O_CREAT, 0666, $DB_BTREE)
    or die "Cannot open file `mytree': $!";

while (($k, $v) = each %hash) {     # Произвести обход.
    print "$k => $v\n";
}
```

Привязать массив к обычному текстовому файлу:

```
tie(@lines, "DB_File", $textfile, O_RDWR|O_CREAT, 0666, $DB_RECNO)
    or die "Cannot open textfile $textfile: $!";

# Записать в файл несколько строк, перекрыв прежнее содержимое.
$lines[0] = "first line";
$lines[1] = "second line";
$lines[2] = "third line";

push @lines, "penult", "last";  # Добавить к файлу две строки.
$wc = scalar @lines;            # Подсчитать строки в файле.
$last = pop @lines;             # Удалить и извлечь последнюю строку.
```

Модуль DB_File предоставляет связанный доступ к базе данных Berkeley DB.[1] Функция tie по умолчанию дает стандартную базу данных в стиле DBM с некоторыми возможностями, которые не предоставляет ни одна другая библиотека DBM: нет ограничений на размер ключей или значений, а данные хранятся в формате, независимом от порядка байтов.

Второй механизм tie использует двоичные деревья, предоставляя действительный файл ISAM (с индексно-последовательным методом доступа), т. е. хеш, ключи которого автоматически упорядочиваются – по умолчанию в алфавитном порядке, но который может конфигурироваться пользователем.

Третий механизм tie привязывает массив к файлу с записями (по умолчанию строками текста), так что изменения в массиве автоматически отражаются на диске. Стандартный интерфейс удовлетворяет версии 1.x Berkeley DB. Те, кому нужны новые функции, доступные в Berkeley DB 2.x или 3.x, могут воспользоваться модулем CPAN BerkeleyDB.

Начиная с версии 2.x, Berkeley DB имеет внутреннюю поддержку блокировки; в более ранних версиях ее нет. В разделе «Блокировка файлов» главы 16 описывается надежный способ блокировки любых файлов баз данных посредством flock с файлом семафора.

---

[1] При условии, что эта библиотека установлена. Если нет, то можно достаточно легко скомпилировать и установить ее.

Dumpvalue

```
use Dumpvalue;

Dumpvalue->new->dumpValue($store);
```

При использовании с примером из `Class::Struct` будет выведено:

```
'Shoppe::addrs' => ARRAY(0x9c1d4)
   0  'Miskatonic University'
   1  'Innsmouth, Mass.'
'Shoppe::boss' => Manager=HASH(0xa5754)
   'Manager::name' => 'Prof L. P. Haitch'
   'Manager::salary' => 'madness'
   'Manager::started' => 'Sat Apr  1 16:18:13 2000'
'Shoppe::owner' => 'Abdul Alhazred'
'Shoppe::stock' => HASH(0xfdbb4)
   'books' => 208
   'charms' => 3
   'potions' => 'none'
```

Это еще один модуль, помогающий выводить сложные данные. Он предназначен не столько для пересылки данных, сколько для их красивого вывода. Он применяется в команде отладчика Perl `x`. В этом качестве он предоставляет огромное количество опций для управления форматом выдачи. Он также обеспечивает удобный доступ к таблицам символов пакетов Perl для дампа содержимого целого пакета.

English

```
use English;

# Использовать имена в стиле awk.
$RS = '';                    # вместо $/
while (<>) {
    next if $NR < 10;        # вместо $.
    ...
}

# То же, но более коболисто.
$INPUT_RECORD_SEPARATOR = '';
while (<>) {
    next if $INPUT_LINE_NUMBER < 10;
    ...
}
```

Модуль `English` предоставляет громоздкие псевдонимы встроенных переменных для тех, кто любит много вводить с клавиатуры и испытывает физическую неприязнь к небуквенным идентификаторам (а также физическую тягу к клавише <Caps Lock>). Как и при всяком импорте, эти псевдонимы до-

ступны только в текущем пакете. Эти переменные остаются доступными под своими действительными именами. Например, после загрузки этого модуля те, кому не нравится $$ или $PROGRAM_NAME, или кого тошнит от $0, могут использовать $PID. У некоторых переменных несколько псевдонимов. Полное описание всех встроенных переменных вместе с их псевдонимами из English см. в главе 28 «Специальные переменные».

Errno

```
use Errno;
unless (open(FH, $pathname)) {
    if ($!{ENOENT}) {  # Нам не нужен для этого импорт!
        warn "$pathname does not exist\n";
    }
    else {
        warn "open failed on `$pathname': $!";
    }
}

use Errno qw(EINTR Eio :POSIX);
if ($! == ENOENT) { .... }
```

Модуль Errno делает доступными символические имена значений статуса завершения при неудаче системного вызова, но по умолчанию не экспортирует их. В модуле один тег для экспорта, ":POSIX", экспортирующий только те символы, которые определены в стандарте POSIX 1003.1. Модуль также превращает в магическую глобальную переменную %!, используя tie. Можно передавать в качестве индекса в хеш %! любой допустимый номер ошибки в системе, не только взятый из POSIX, и значение будет истинным, только если оно соответствует текущей ошибке.

Exporter

Внутри файла *MyModule.pm*:

```
package MyModule;

use strict;
use Exporter;

our $VERSioN = 1.00;           # Или выше...
our @ISA = qw(Exporter);
our @EXPORT      = qw(f1 %h);  # Символы, импортируемые по умолчанию.
our @EXPORT_OK   = qw(f2 f3);  # Символы, импортируемые только по требованию
our %EXPORT_TAGS = (           # Соответствие для :сокращений.
    a => [qw(f1 f2 f3)],
    b => [qw(f2 %h)],
);
```

```
# Здесь ваш код.

1;
```

Из программы или другого модуля, взаимодействующих с вашим:

```
use MyModule;                    # Импортировать все в @EXPORT.
use MyModule ();                 # Загрузить модуль, никакого импорта.
use MyModule "f1", "f2", "%h";   # Две подпрограммы и переменная.
use MyModule qw(:DEFAULT f3);    # Все в @EXPORT + одна подпрограмма.
use MyModule "f4";               # Фатально, потому что f4 не экспортирована.
```

Когда кто-нибудь посредством объявления `use` загружает ваш модуль, он вызывает метод `import` этого модуля для загрузки нужных символов в пакет вызывающего. Ваш модуль (тот, который производит экспорт) может определить метод `import` любым угодным способом, но стандартный путь состоит в наследовании метода из модуля класса `Exporter`. Это то, что делает приведенный выше код.

Модуль `Exporter` служит базовым классом для модулей, которые хотят организовать собственный экспорт. Как ни странно, объектно-ориентированные модули обычно не используют `Exporter`, поскольку обычно они ничего не экспортируют (вызовы методов не требуется экспортировать). Однако к самому модулю `Exporter` доступ организован в стиле ООП из-за установленного массива `@ISA`, как в нашем примере. Когда другая программа или модуль выполняет `use` для какого-либо модуля, метод `import` вызывается в этом модуле как метод класса: `MyModule->import(`*`LIST`*`)`. Однако поскольку метод `import` в нем не определен, то автоматически используется метод `Exporter::import` через наследование.

Массив `@EXPORT` модуля содержит список символов (функций и даже переменных), которые вызывающий код автоматически импортирует чистой командой `use`. Массив `@EXPORT_OK` содержит символы, которые можно импортировать, если специально указать их имена. Выполняется сверка с номером `$VERSION`, если в команде `use` запрашивается конкретная версия модуля. Доступны многие и многие другие возможности. См. главу 11 «Модули», а также электронную страницу руководства для модуля `Exporter`.

Fatal

Модуль `Fatal` делает отказ функций более зрелищным. Он заменяет функции, которые обычно возвращают при неуспехе «ложь», оболочками, которые возбуждают исключительную ситуацию, когда действительная функция возвращает «ложь». Благодаря этому можно безопасно применять эти функции, не проверяя возвращаемые ими значения при каждом вызове.

Как функции, определяемые пользователем, так и встроенные, могут быть заключены в оболочку, исключая те встроенные, которые нельзя выразить прототипами. Попытка замещения незамещаемых встроенных функций при-

водит к исключительной ситуации. В их число входят system, print, printf, exec, split, grep и map — или, более обобщенно, любая *FUNC*, для которой "CORE::*FUNC*" возвращает значение «ложь», в том числе сама функция prototype.

Если в списке импорта появляется символ :void, указанные далее функции могут возбуждать исключительную ситуацию только при вызове функции в пустом контексте — т. е. когда возвращаемое значение игнорируется. (Будьте внимательны с последней командой подпрограммы.) Например:

```
use Fatal qw(:void open close);

# open проверяется, чтобы при неудаче не  создавалась исключительная ситуация.
if (open(FH, "< /nonesuch") { warn "no /nonesuch: $!"; }

# close не проверяется, поэтому при неудаче создавалась исключительная ситуация.
close FH;
```

Fcntl

```
use Fcntl;              # Импортировать стандартные константы fcntl.h.
use Fcntl ":flock";     # Импортировать константы LOCK_*.
use Fcntl ":seek";      # Импортировать SEEK_CUR, SEEK_SET, SEEK_END.
use Fcntl ":mode";      # Импортировать S_* stat проверочные константы.
use Fcntl ":Fcompat";   # Импортировать константы F*.
```

Модуль Fcntl предоставляет константы, используемые с различными встроенными функциями Perl. В импортируемую по умолчанию группу входят константы F_GETFL и F_SETFL для fcntl, SEEK_SET и SEEK_END для seek и sysseek, а также O_CREAT и O_EXCL для sysopen. В число поддерживаемых тегов импорта входят ":flock" для доступа к константам LOCK_EX, LOCK_NB, LOCK_SH и LOCK_UN для flock; ":mode" для доступа к константам *sys/stat.h*, таким как S_IRUSR и S_ISFIFO; ":seek" для доступа к трем аргументам seek и sysseek; ":Fcompat" для доступа к старомодным символам с "F", а не "F_" в начале, таким как FAPPEND, FASYNC и FNONBLOCK. См. электронную документацию к модулю Fcntl и документацию операционной системы по соответствующим системным вызовам *fcntl*(2), *lseek*(2), *open*(2) и *stat*(2).

File::Basename

```
use File::Basename;

$fullname = "/usr/local/src/perl-5.6.1.tar.gz";

$file = basename($fullname);
# file="perl-5.6.1.tar.gz"

$dir = dirname($fullname);
# dir="/usr/local/src"
# dir="/usr/local/src/" file="perl-5" ext=".6.1.tar.gz"
```

```
 ($file,$dir,$ext) = fileparse($fullname, qr/\.[^.]*/);
# dir="/usr/local/src/" file="perl-5.6.1.tar" ext=".gz"

 ($file,$dir,$ext) = fileparse($fullname, qr/\.\D.*/);
# dir="/usr/local/src/" file="perl-5.6.1" ext=".tar.gz"

 ($file,$dir,$bak) = fileparse("/tmp/file.bak",
                       qr/~+$/, qr/\.(bak|orig|save)/)
# dir="/tmp/" file="file" ext=".bak"

 ($file,$dir,$bak) = fileparse("/tmp/file~",
                       qr/~+$/, qr/\.(bak|orig|save)/)
# dir="/tmp/" file="file" ext="~"
```

Модуль `File::Basename` предоставляет функции для разбора имен путей на отдельные компоненты. Функция `dirname` извлекает часть, относящуюся к каталогу, а функция `basename` – часть, не являющуюся каталогом. Более искусная функция `fileparse` разбирает полное имя пути на имя каталога, имя файла и суффикс; необходимо передать список регулярных выражений, описывающих нужные вам суффиксы. Приведенные выше примеры описывают, как шаблоны для суффиксов влияют на результат. По умолчанию эти функции анализируют имя пути в соответствии с соглашениями, принятыми для текущей платформы. Функция `fileparse_set_fstype` выбирает правила анализа для другой платформы, например, `fileparse_set_fstype("VMS")` для анализа имен по правилам VMS, даже при выполнении на системах, отличных от VMS.

File::Compare

```
use File::Compare;

printf "fileA and fileB are %s.\n",
    compare("fileA","fileB") ? "different" : "identical";

use File::Compare 'cmp';
sub munge($) {
    my $line = $_[0];
    for ($line) {
        s/^\s+//;   # Убрать ведущие пробельные символы.
        s/\s+$//;   # Убрать замыкающие пробельные символы.
    }
    return uc($line);
}

if (not cmp("fileA", "fileB", sub {munge $_[0] eq munge $_[1]} ) {
    print "fileA and fileB are kinda the same.\n";
}
```

Модуль `File::Compare` предоставляет одну функцию – `compare`, – которая сравнивает содержимое двух передаваемых ей файлов. Она возвращает `0`, если

файлы содержат одинаковые данные, 1, если они содержат разные данные, и -1, если произошла ошибка при обращении к указанным файлам. Если в качестве третьего аргумента передается ссылка на подпрограмму, то эта функция многократно вызывается для определения эквивалентности каждой пары строк. Для совместимости с программой *cmp*(1) можно явно импортировать функцию как cmp. (Это не оказывает влияния на бинарный оператор cmp.)

File::Copy

```
use File::Copy;

copy("/tmp/fileA", "/tmp/fileA.orig") or die "copy failed: $!";
copy("/etc/motd", *STDOUT)            or die "copy failed: $!";
move("/tmp/fileA", "/tmp/fileB")      or die "move failed: $!";

use File::Copy qw/cp mv/;             # Get normal Unix names.
cp "/tmp/fileA", "/tmp/fileA.orig"    or die "copy failed: $!";
"/tmp/fileA", "/tmp/fileB"         or die "move failed: $!";
```

Модуль File::Copy экспортирует две функции – copy и move, – которые, соответственно, копируют или переименовывают первый аргумент во второй, аналогично вызову команд Unix *cp*(1) и *mv*(1) (имена, которые можно использовать при явном импорте). Функция copy принимает в качестве аргументов и указатели файлов. Эти функции возвращают значение «истина» в случае успеха и «ложь» при неудаче, устанавливая надлежащим образом $! ($OS_ERROR). (К сожалению, невозможно узнать, относится ли нечто вроде «Permission denied» к первому файлу или второму.) Эти функции позволяют достичь некоторого компромисса между удобством и точностью. Они не поддерживают многочисленные опции и оптимизации, имеющиеся в *cp*(1) и *mv*(1), например, рекурсивное копирование, автоматическое резервное копирование, сохранение исходных временных меток и информации о владении, а также интерактивного подтверждения. Доступ к каким-либо из указанных возможностей лучше всего получать посредством вызова версий этих команд для конкретной платформы.[1] Просто имейте в виду, что не все системы поддерживают одни и те же команды или опции для них.

```
system("cp -R -pi /tmp/dir1 /tmp/dir2") == 0
    or die "external cp command status was $?";
```

File::Find

```
use File::Find;

# Вывести все каталоги ниже текущего.
find sub { print "$File::Find::name\n" if -d }, ".";
```

[1] Или получить версии PPT (Perl Power Tools), если платформа «конфликтует» с инструментальными средствами.

```
# Вычислить общий объем памяти, используемый всеми файлами в перечисленных
каталогах.
@dirs = @ARGV ? @ARGV : ('.');
my $sum = 0;
find sub { $sum += -s }, @dirs;
print "@dirs contained $sum bytes\n";

# Изменить режим по умолчанию для прохода через символические ссылки
# и сначала посетить подкаталоги
find { wanted => \&myfunc, follow => 1, bydepth => 1 }, ".";
```

Функция `find` модуля `File::Find` рекурсивно обходит каталоги. Ее первый аргумент должен быть ссылкой на функцию, а все последующие аргументы должны быть каталогами. Функция вызывается с каждым именем файла из перечисленных каталогов. Внутри этой функции переменная `$_` устанавливается равной базовому имени текущего файла, а текущий рабочий каталог процесса устанавливается по умолчанию в его каталог. В переменную пакета `$File::Find::name` заносится полное имя пути для посещаемого файла. В альтернативном соглашении по вызову в качестве первого аргумента принимается ссылка на хеш, содержащий заданные опции, в том числе `"wanted"`, `"bydepth"`, `"follow"`, `"follow_fast"`, `"follow_skip"`, `"no_chdir"`, `"untaint"`, `"untaint_pattern"` и `"untaint_skip"`, которые полностью описываются в электронной документации. Этот модуль используется также стандартной программой-транслятором *find2perl*(1), которая поставляется с Perl.

File::Glob

```
use File::Glob ':glob';      # Заместить встроенную glob.
@list = <*.[Cchy]>;          # Теперь используется glob из POSIX, не csh.

use File::Glob qw(:glob csh_glob);
@sources = bsd_glob("*.{C,c,h,y,pm,xs}", GLOB_CSH);
@sources = csh_glob("*.{C,c,h,y,pm,xs}");  # (то же)

use File::Glob ':glob';
# call glob with extra arguments
$homedir = bsd_glob('~jrhacker', GLOB_TILDE | GLOB_ERR);
if (GLOB_ERROR) {
    # Произошла ошибка при разыменовании исходного каталога.
}
```

Функция `bsd_glob` модуля `File::Glob` реализует программу *glob*(3) из библиотеки C. Необязательный второй аргумент содержит флаги, управляющие дополнительными свойствами поиска. Тег импорта `:glob` импортирует как функцию, так и необходимые флаги.

В модуле также реализована функция `csh_glob`. Это ее в действительности вызывают встроенный оператор Perl `glob` и операторы *`GLOBPAT`*. Вызов `csh_glob` в основном подобен такому вызову `bsd_glob`:

```
bsd_glob(@_ ? $_[0] : $_,
    GLOB_BRACE | GLOB_NOMAGIC | GLOB_QUOTE | GLOB_TILDE);
```

Если импортируется тег `:glob`, то при всех вызовах встроенных операторов поиска файлов в текущем пакете будет вызываться функция `bsd_glob`, а не `csh_glob`. Одна из причин именно такой реализации состоит в том, что `bsd_glob` правильно обрабатывает шаблоны, содержащие пробельные символы, а `csh_glob` обрабатывает их в историческом стиле. В старых сценариях писали `<*.c *.h>` для поиска файлов обоих типов. Ни ту ни другую функцию, однако, не беспокоят пробелы в фактических именах файлов.

Функция `bsd_glob` принимает аргумент, содержащий шаблон для поиска файлов (не шаблон регулярного выражения) и необязательный аргумент флагов. Поиск имен файлов, начинающихся с точки, выполняется только по специальному запросу. На возвращаемое значение оказывают влияние флаги во втором аргументе, которые можно логически складывать поразрядно:[1]

`GLOB_BRACE`

Предварительная обработка строки для расширения строк `{pat,pat,...}`, как делает *csh*(1). Шаблон `{}` не расширяется по историческим причинам, в основном для облегчения ввода шаблонов *find*(1).

`GLOB_CSH`

Синоним для `GLOB_BRACE | GLOB_NOMAGIC | GLOB_QUOTE | GLOB_TILDE`.

`GLOB_ERR`

Возвратить ошибку, если `bsd_glob` обнаружит каталог, который не может открыть или прочесть. Обычно `bsd_glob` пропускает ошибку и ищет дальнейшие совпадения.

`GLOB_MARK`

Возвращать значения, являющиеся каталогами, с косой чертой в конце.

`GLOB_NOCASE`

По умолчанию в именах файлов учитывается регистр; этот флаг делает `bsd_glob` нечувствительной к регистру. (См. ниже об исключениях для систем типа MS-DOS).

`GLOB_NOCHECK`

Если шаблон не соответствует ни одному имени пути, заставляет `bsd_glob` возвратить список, состоящий только из шаблона, как это делает */bin/sh*. Если установлен `GLOB_QUOTE`, он действует в возвращаемом шаблоне.

`GLOB_NOMAGIC`

То же, что `GLOB_NOCHECK`, но шаблон возвращается, только если он не содержит специальных символов `*`, `?` или `[`. Флаг `NOMAGIC` предоставляется для

[1] Из-за ограничений синтаксиса встроенного оператора `glob` для передачи второго аргумента может потребоваться вызов функции `bsd_glob`.

упрощения реализации исторического режима поиска *csh*(1) и, вероятно, нигде больше использоваться не должен.

GLOB_NOSORT

По умолчанию имена маршрутов сортируются в возрастающем порядке (с применением обычного сравнения символов без учета национальных установок). Этот флаг отменяет сортировку и немного увеличивает скорость.

GLOB_QUOTE

Использовать обратный слэш \ для цитирования: каждое вхождение в шаблон обратного слэша, за которым следует символ, заменяется этим символом без особой интерпретации этого символа. (См. ниже об исключениях для систем типа MS-DOS.)

GLOB_TILDE

Разрешить шаблоны, в которых путь начинается с ~*USER*. Если имя пользователя *USER* опущено, то сама тильда (возможно, с последующей косой чертой) представляет исходный каталог текущего пользователя.

Функция bsd_glob возвращает список (возможно, пустой) совпадающих путей, который будет помечен (tainted), если это имеет значение для программы. В случае ошибки GLOB_ERROR будет иметь значение «истина», а $! ($OS_ERROR) будет содержать стандартное системное сообщение об ошибке. При отсутствии ошибки GLOB_ERROR имеет значение «ложь», в противном случае значение GLOB_ABEND или GLOB_NOSPACE. (GLOB_ABEND означает, что bsd_glob была остановлена из-за какой-то ошибки, GLOB_NOSPACE означает, что кончилась память.) Если bsd_glob уже нашла какие-то пути, когда произошла ошибка, она возвращает список имен файлов, найденных к данному моменту, *а также устанавливает* GLOB_ERROR. Обратите внимание, что такая реализация bsd_glob отличается от большинства других тем, что не рассматривает ENOENT и ENOTDIR как условия завершения с ошибкой. Она продолжает работу, несмотря на эти ошибки, если не установлен флаг GLOB_ERR.

Если аргумента с флагами нет, используются значения, установленные в системе по умолчанию, что означает неразличимость одного от другого имен, отличающихся только регистром букв в системах VMS, OS/2, старой Mac OS (но не Mac OS X) и Microsoft (но не тогда, когда Perl компилировался с Cygwin). Если мы передаем какие-либо флаги, но хотим, чтобы осуществлялся такой режим, то должны включить в число флагов GLOB_NOCASE. В любой системе можно изменить установки по умолчанию, импортировав теги :case или :nocase.

В системах типа MS-DOS обратный слэш является допустимым символом – разделителем каталогов.[1] В этом случае применение обратного слэша для ци-

[1] Хотя технически им является и обычная косая черта – по крайней мере, это относится к ядру и системным вызовам; образованность командных оболочек заметно ниже.

тирования символа (через GLOB_QUOTE) конфликтует с использованием ее как разделителя каталогов. Самое хорошее (простое, наиболее переносимое) решение состоит в том, чтобы отвести слэшу роль разделителя каталогов, а для цитирования применять обратный слэш. Однако некоторые пользователи рассчитывают увидеть иное, поэтому обратный слэш (при задании GLOB_QUOTE) предназначен для цитирования только метасимволов поиска [,], {, }, -, ~ и самой себя. Все остальные символы за обратным слэшем проходят без изменений, если удается получить их в результате собственного цитирования обратного слэша в строках Perl. Может потребоваться целых четыре обратных слэша, чтобы найти один в файловой системе. Это совершенное безумие, и даже пользователям MS-DOS следует серьезно подумать о применении обычного слэша. Если уж действительно необходимо использовать обратный слэш, обратитесь к стандартному модулю File::DosGlob, т. к. работа с ним может оказаться привлекательнее, чем поиск файлов в стиле Unix.

File::Spec

```
use File::Spec;     # OO стиль

$path = File::Spec->catfile("subdir", "filename");
    # 'subdir/filename' в Unix, OS2 или Mac OS X
    # 'subdir:filename' в (старых) Apple Macs
    # 'subdir\filename' в Microsoft

$path = File::Spec->catfile("", "dir1", "dir2", "filename");
    # '/dir1/dir2/filename' в Unix, OS2 или Mac OS X
    # ':dir1:dir2:filename' в (старых) Apple Macs
    # '\dir1\dir2\filename' в Microsoft

use File::Spec::Unix;
$path = File::Spec::Unix->catfile("subdir", "filename");
    # 'subdir/filename' (даже при выполнении в системах, отличных от Unix)

use File::Spec::Mac;
$path = File::Spec::Mac->catfile("subdir", "filename");
    # 'subdir:filename'

use File::Spec::Win32;
$path = File::Spec::Win32->catfile("subdir", "filename";)
    # 'subdir\filename'

# Использовать интерфейс функций.
use File::Spec::Functions;
$path = catfile("subdir", "filename");
```

Семейство модулей File::Spec позволяет строить маршруты с учетом каталогов и имен файлов без жесткого кодирования разделителей каталогов, специфичных для платформ. В число поддерживаемых систем входят Unix, VMS, Mac и Win32. Во всех этих модулях есть метод класса catfile, который конкатенирует составляющие маршрута, используя разделитель для ука-

занной платформы. Модуль File::Spec возвращает различные результаты в зависимости от текущей платформы. Остальные возвращают результаты, специфичные для этой платформы. Модуль File::Spec::Functions предоставляет интерфейс функций.

File::stat

```
use File::stat;
$st = stat($file)      or die "Can't stat $file: $!";
if ($st->mode & 0111 and $st->nlink > 1)) {
    print "$file is executable with many links\n";
}

use File::stat ":FIELDS";
stat($file)            or die "Can't stat $file: $!";
if ($st_mode & 0111 and $st_nlink > 1) ) {
    print "$file is executable with many links\n";
}

@statinfo = CORE::stat($file);    # Доступ к замещенной встроенной функции.
```

Модуль File::stat предоставляет интерфейс методов к встроенным функциям Perl stat и lstat, заменяя их версиями, возвращающими объект File::stat (или undef при неудаче). У этого объекта есть методы, возвращающие одноименные поля из структуры обычного системного вызова *stat*(2), а именно dev, ino, mode, nlink, uid, gid, rdev, size, atime, mtime, ctime, blksize и blocks. Можно также импортировать поля структуры в собственное пространство имен в качестве обычных переменных посредством тега импорта ":FIELDS". (При этом все равно замещаются встроенные stat и lstat.) Эти поля выглядят как скалярные переменные, имена которых содержат "st_" перед именем поля. То есть переменная $st_dev соответствует методу $st->dev.

File::Temp

```
use File::Temp qw(tempfile tempdir);

$dir = tempdir(CLEANUP => 1);
($fh, $filename) = tempfile(DIR => $dir);
($fh, $filename) = tempfile($template, DIR => $dir);
($fh, $filename) = tempfile($template, SUFFIX => ".data");
$fh = tempfile();

use File::Temp ':mktemp';

 ($fh, $filename) = mkstemp("tmpfileXXXXX");
($fh, $filename) = mkstemps("tmpfileXXXXXX", $suffix);
$tmpdir = mkdtemp($template);
$unopened_file = mktemp($template);
```

Новый модуль File::Temp, появившийся в версии Perl 5.6.1, предоставляет удобные функции для безопасного создания и открытия временных файлов. Лучше воспользоваться этим модулем, чем пытаться самостоятельно выбрать временный файл. Это поможет избежать повторения ошибок, совершенных ранее. Этот модуль оберегает от различных тонких ситуаций гонки, а также от опасности использования каталогов, в которые могут вести запись другие; см. главу 23 «Защита данных». Функция tempfile возвращает и указатель файла, и его имя. Надежнее всего работать с уже открытым указателем файла и игнорировать имя файла полностью (за исключением, возможно, сообщений об ошибках). Если закрыть файл, то он автоматически удаляется. Для совместимости с библиотекой C тег импорта :mktemp предоставляет доступ к функциям, имена которых знакомы программирующим на C, но помните, что имена файлов всегда менее безопасны, чем указатели файлов.

FileHandle

```
use FileHandle;

$fh = new FileHandle;
if ($fh->open("< file")) {
    print $line while defined($line = $fh->getline);
    $fh->close;
}

$pos = $fh->getpos;     # как tell()
$fh->setpos($pos);      # как seek()

 ($readfh, $writefh) = FileHandle::pipe();

autoflush STDOUT 1;
```

Модуль FileHandle в основном служит тому, чтобы преобразовать переменные пунктуации Perl в длинные вызовы, более похожие на объектные. Он поставляется для совместимости с ранними версиями, но в действительности служит интерфейсом для нескольких более специфических модулей, таких как io::Handle и io::File.[1] Его лучшее свойство – обеспечение низкоуровневого доступа к некоторым редким функциям из библиотеки C (*clearerr*(3), *fgetpos*(3), *fsetpos*(3) и *setvbuf*(3)).

Переменная	Метод
$|	autoflush
$,	output_field_separator
$\	output_record_separator

[1] Он загружает столько кода, что обойдется примерно в мегабайт памяти.

(*продолжение*)

Переменная	Метод
$/	input_record_separator
$.	input_line_number
$%	format_page_number
$=	format_lines_per_page
$-	format_lines_left
$~	format_name
$^	format_top_name
$:	format_line_break_characters
$^L	format_formfeed

Вместо:

```
$ofh = select(HANDLE);
$~ = 'SomeFormat';
$| = 1;
select($ofh);
```

можно просто сказать:

```
use FileHandle;
HANDLE->format_name('SomeFormat');
HANDLE->autoflush(1);
```

В настоящее время три метода (output_field_separator, output_record_separator и input_record_separator) только делают вид, что работают с отдельным указателем: установка их для одного указателя оказывает воздействие на все указатели файлов. Поэтому они поддерживаются только как методы класса, а не отдельного указателя. Возможно, это ограничение будет снято.

Чтобы получить указатель файла с лексической областью видимости, вместо самооживления указателя файла:

```
open my $fh, "< somefile"
    or die "can't open somefile: $!";
```

можно сказать:

```
use FileHandle;
my $fh = FileHandle->new("< somefile")
    or die "can't open somefile: $!";
```

FileHandle наследует IO::File, который наследует IO::Handle и IO::Seekable. Практически для всех функций модуля есть более эффективный доступ посредством основных простых вызовов Perl, за исключением следующих, которые могут быть реализованы не на всех платформах, отличных от Unix.

HANDLE->blocking(*EXPR*)

При вызове с аргументом включает неблокирующий ввод-вывод, если аргумент имеет значение «ложь», и выключает неблокирование (т. е. включает блокирование), если аргумент имеет значение «истина». Метод возвращает предшествующее установленное значение (которое остается действующим, если аргумент не задан). При ошибке blocking устанавливает $! и возвращает undef. Это можно сделать непосредственно с помощью fcntl, но работать с интерфейсом FileHandle значительно проще.

HANDLE->clearerr

Вызывает функцию библиотеки C *clearerr*(3) для сброса внутренних индикаторов указателя состояния конца файла и ошибки.

HANDLE->error

Вызывает функцию библиотеки C *ferror*(3) для проверки индикатора ошибки заданного указателя файла. Возвращает истину, если указатель установлен. Надежный способ сброса индикатора предоставляет только метод clearerr. (Некоторые системы сбрасывают его также при вызове оператора seek.)

HANDLE->formline(*PICTURE*, *LIST*)

Это то же самое, что сохранить переменную аккумулятора ($^A), вызвать функцию formline с заданными *PICTURE* и *LIST* и вывести полученное содержимое аккумулятора по заданному указателю, а затем восстановить прежнее содержимое аккумулятора. Вот, например, как вывести длинную текстовую переменную с автоматическим переносом слов по колонке 72:

```
use FileHandle;
STDOUT->formline("^" . ("<" x 72) . "~~\n", $long_text);
```

HANDLE->getpos

Вызывает функцию библиотеки C *fgetpos*(3), предоставляя альтернативный для tell интерфейс. В некоторых (не-UNIX) системах возвращаемое значение может быть сложным объектом, а getpos и setpos могут быть единственным переносимым способом изменения позиции в текстовом потоке.

FileHandle->new_tmpfile

Вызывает функцию библиотеки C *tmpfile*(3), чтобы создать новый временный файл, открытый в режиме чтения-записи, и возвращает указатель на этот поток. В некоторых системах допускается анонимность временного файла – т. е. после создания для него выполняется unlink, но он остается открытым. Эту функцию или POSIX::tmpnam, описанную в модуле POSIX, нужно использовать для безопасного создания временного файла, не создавая тонких, но серьезных проблем безопасности, возникающих в ситуации гонок. В версии Perl 5.6.1 предпочтительным интерфейсом является теперь модуль File::Temp.

HANDLE->setbuf(*BUFFER*)

Вызывает функцию библиотеки C *setbuf*(3) с заданной переменной *BUFFER*. Чтобы задать небуферизованный вывод, нужно передать undef. Переменную, используемую в качестве буфера в setbuf или setvbuf, ***нельзя никоим образом модифицировать***, пока не будет закрыт указатель или пока снова не будут вызваны setbuf или setvbuf. В противном случае может произойти нарушение памяти, и вам будет нехорошо.

HANDLE->setpos(*EXPR*)

Вызывает функцию библиотеки C *fsetpos*(3), предоставляя альтернативный для seek интерфейс. Аргументом может быть только значение, возвращенное getpos, описанной выше.

HANDLE->setvbuf(*BUFFER*, *TYPE*, *SIZE*)

Вызывает функцию библиотеки C *setvbuf*(3) с заданным *BUFFER*. Для поля *TYPE* можно использовать стандартные константы библиотеки C _ioNBF (без буферизации), _ioLBF (буферизация строк) и _ioFBF (полная буферизация), если их явно импортировать. См. предупреждение, сделанное для setbuf.

HANDLE->sync

Вызывает функцию библиотеки C *fsync*(3) для синхронизации состояния файла в памяти и на физическом носителе. Обратите внимание, что sync действует не с указателем файла, а с его дескриптором, поэтому данные в буферах не будут синхронизированы, если сначала не очистить буферы.

HANDLE->untaint

Помечает указатель файла или каталог как служащие источником немеченых (untainted) данных. При работе в режиме меченых данных (см. главу 23 «Защита данных») данные, полученные из внешних файлов, считаются не заслуживающими доверия. Не вызывайте этот метод, не подумав: это сведет на нет лучшие усилия Perl защитить вас от вас самого.

Getopt::Long

Если программа говорит:

```
use Getopt::Long;
GetOptions("verbose"  => \$verbose,
           "debug"    => \$debug,
           "output=s" => \$output);
```

то из командной строки ее можно вызывать так:

```
% prog - -verbose другие аргументы
% prog - -debug другие аргументы
% prog -v -d другие аргументы
% prog - -output=somefile другие аргументы
% prog -o somefile другие аргументы
```

Модуль Getopt::Long предоставляет функцию GetOptions для обработки ключей командной строки с длинными именами. Он включает поддержку сокращенных ключей, типизированных аргументов, например, булевых, строковых, целых и действительных чисел, массивов для повторения опций, пользовательских процедур проверки допустимости, согласованности с POSIX против обработки в стиле FSF, опций, не чувствительных к регистру, и обычное связывание коротких опций – вот лишь некоторые из этого рога изобилия функций. Если возможности этого модуля чрезмерны для вас, обратите внимание на более традиционный модуль Getopt::Std, описываемый ниже. Если его возможностей окажется недостаточно, возьмите в CPAN модуль Getopt::Declare, предоставляющий более декларативный синтаксис задания опций.

Getopt::Std

```
use Getopt::Std;
```

Можно вызывать getopt и getopts с глобальными переменными:

```
our ($opt_o, $opt_i, $opt_f);
getopt('oif');            # -o, -i и -f принимают аргументы.
                          # Устанавливает глобальные переменные $opt_*.
getopts('oif:');          # Теперь -o & -i булевы; -f принимает аргумент.
                          # Как побочный эффект устанавливает глобальные $opt_* .
```

Либо можно использовать их с закрытым хешем опций:

```
my %opts;                 # Сюда поместим результаты.
getopt('oif', \%opts);    # Все три принимают аргументы.
getopts('oif:', \%opts);  # Теперь -o и -i - булевы флаги
                          # и только -f принимает аргумент.
```

Модуль Getopt::Std предоставляет две функции, getopt и getopts, которые помогают выделять в аргументах командной строки односимвольные опции. Функция getopts более полезна, поскольку позволяет указать, что некоторые опции принимают аргументы, а другие – нет, тогда как getopt предполагает, что все опции принимают аргументы. Передавая getopts букву, после которой стоит двоеточие, мы указываем, что эта опция принимает аргумент, в противном случае ожидается булев флаг. Поддерживается обычное склеивание опций. Порядок не имеет значения, поэтому можно группировать опции, не принимающие аргументы. Опции, которые принимают аргумент, должны быть последними в группе или стоять отдельно, а их аргумент может следовать непосредственно за опцией или как следующий аргумент программы. Для примера getopts, приведенного выше, следующие вызовы эквивалентны:

```
% prog -o -i -f TMPFILE другие аргументы
% prog -o -if TMPFILE другие аргументы
% prog -io -fTMPFILE другие аргументы
```

```
% prog -iofTMPFILE другие аргументы
% prog -oifTMPFILE другие аргументы
```

IO::Socket

```
use io::Socket;
```

В качестве клиента:

```
$socket = new io::Socket::INET (PeerAddr => $remote_host,
                                PeerPort => $remote_port,
                                Proto    => "tcp",
                                Type     => SOCK_STREAM)
    or die "Can't connect to $remote_host:$remote_port : $!\n";
$socket = io::Socket::INET->new("$remote_host:$remote_port");
                              # "localhost:80", for example.

print $socket "data\n";
$line = <$socket>;
```

В качестве сервера:

```
$server = io::Socket::INET->new(LocalPort => $server_port,
                                Type      => SOCK_STREAM,
                                Reuse     => 1,
                                Listen    => 10 )  # or SOMAXCONN
    or die "Can't be a TCP server on port $server_port : $!\n";

while ($client = $server->accept()) {
    # $client является новым соединением
    $request = <$client>;
    print $client "answer\n";
    close $client;
}

# Сделаем простую функцию соединения TCP, которая возвращает
# указатель файла для использования в простых программах клиентов.
sub tcp_connect {
    my ($host, $service) = @_;
    require io::Socket;
    return  io::Socket::INET->new(join ":", $host, $service);
}
my $fh    = tcp_connect("localhost", "smtp");  # with scalar
local *FH = tcp_connect("localhost", "smtp");  # with handle
```

Модуль `io::Socket` предоставляет доступ к сокетам на более высоком уровне, чем простой модуль `Socket`. Можно обращаться к нему в объектно-ориентированном стиле, хотя это не обязательно, поскольку возвращаемые значения являются действительными указателями файлов и как таковые могут использоваться, что показано выше на примере функции `tcp_connect`. Этот модуль наследует методы у `io::Handle` и сам загружает `io::Socket::INET` и `io::So-`

cket::UNIX. См. другие интересные функции в описании модуля FileHandle. Описание работы с сокетами приведено в главе 16.

IPC::Open2

```
use IPC::Open2;

local(*HIS_OUT, *HIS_IN);  # Создать, если нужно, локальные указатели.

$childpid = open2(*HIS_OUT, *HIS_IN, $program, @args)
    or die "can't open pipe to $program: $!";
print HIS_IN "here's your input\n";
$his_output = <HIS_IN>;
close(HIS_OUT);
close(README);
waitpid($childpid, 0);
```

Единственная экспортируемая модулем IPC::Open2 функция — open2 — запускает другую программу и предоставляет доступ к этой команде по чтению и записи. Первые два аргумента должны быть допустимыми указателями файлов (или пустыми переменными, в которые можно поместить самогенерируемые указатели файлов). Остальные аргументы — это программа и ее аргументы, которые, если передаются раздельно, не подвергаются интерполяции оболочкой. Этот модуль не убирает дочерние процессы после выхода. За исключением коротких программ, в которых допустимо разрешить операционной системе позаботиться об этом, программист должен делать это сам. Обычно это всего лишь вызов waitpid $pid, 0 по завершении работы с этим дочерним процессом. Невыполнение этого может привести к накоплению погибших процессов («зомби»).

На практике этот модуль не очень хорошо работает со многими программами из-за того, как осуществляется буферизация в стандартных библиотеках C ввода-вывода. Однако при наличии исходного кода обеих программ можно легко преодолеть это ограничение, очищая выходные буферы чаще, чем это делается по умолчанию. Если этого не сделать, то программы будут с раздражающей скупостью копить выводимые данные. Другой потенциальной опасностью является взаимоблокировка: если оба процесса одновременно читают и ни один не пишет, то программа повиснет. Более подробное изложение см. в главе 16.

IPC::Open3

```
use IPC::Open3;

local(*HIS_IN, *HIS_OUT, *HIS_ERR);

$childpid = open3(*HIS_IN, *HIS_OUT, *HIS_ERR, $cmd, @args);
print HIS_IN "stuff\n";
close(HIS_IN);            # Потомок видит конец файла.
```

```
@outlines = <HIS_OUT>;    # Читать до EOF.
@errlines = <HIS_ERR>;    # XXX: опасность блокировки при большом объеме
print "STDOUT:\n", @outlines, "\n";
print "STDERR:\n", @errlines, "\n";
close HIS_OUT;
close HIS_ERR;
waitpid($childpid, 0);
if ($?) {
    print "That child exited with wait status of $?\n";
}
```

Модуль IPC::Open3 работает аналогично IPC::Open2 (последний реализован с помощью первого), за исключением того, что open3 предоставляет доступ к указателям стандартного устройства ввода, стандартного устройства вывода и стандартного устройства ошибок запускаемой программы. Действуют те же предупреждения, что и в случае open2 (см. предыдущий раздел), плюс некоторые дополнительные. Порядок аргументов в open3 отличается от порядка в open2. Вместо передачи указателя для чтения в первом и указателя для записи во втором теперь все наоборот. Кроме того, в open3 опасность взаимоблокировки еще выше. Если попытаться осуществить чтение за концом файла на одном из двух выходных указателей потомка, в то время как есть большой объем выдачи на другом, равный процесс блокируется и оказывается зависшим. Чтобы избежать этого, используйте формат select с четырьмя аргументами или стандартный модуль io::Select. Дополнительные сведения см. в главе 16.

Math::BigInt

```
use Math::BigInt;
$i = Math::BigInt->new($string);

use Math::BigInt ':constant';
print 2**200;
```

Этот код выводит:

```
+1606938044258990275541962092341162602522202993782792835301376
```

Модуль Math::BigInt предоставляет объекты, представляющие целые числа с произвольной точностью и перегруженные арифметические операторы. Эти объекты создаются с помощью конструктора new или, в лексической области видимости, путем импорта особого значения ":constant", после чего все числовые литералы до конца лексической области видимости рассматриваются как объекты Math::BigInt. Реализованы все обычные операторы для целых чисел, а в версии Perl 5.6 и поразрядные логические операторы. В текущей реализации этот модуль нельзя назвать быстрым, как молния, но в будущем это может измениться. (Хотелось бы узнать, с какой скоростью *вы* вычислите в уме 2**200.)

Math::Complex

```
use Math::Complex;

$z = Math::Complex->make(5, 6);
$z = cplx(5, 6);          # то же, но короче
$t = 4 - 3*i + $z;        # стандартные действия с комплексными числами
print "$t\n";             # выводит 9+3i

print sqrt(-9), "\n";     # выводит 3i
```

Модуль Math::Complex предоставляет объекты комплексных чисел с перегруженными операторами. Это числа, имеющие действительную и мнимую части, например, те, которые являются корнями четной степени из отрицательных чисел, как показано выше. Помимо арифметических операторов многие встроенные математические функции (к которым относятся abs, log, sqrt, sin, cos и atan2) замещены версиями, понимающими комплексные числа. Имеются также функции Re и Im, выделяющие действительную и мнимую части своих комплексных аргументов, а также полный набор расширенных тригонометрических функций, например tan, asin, acos, sinh, cosh и tanh. Модуль также экспортирует константу i, которая, как можно догадаться, содержит величину *i*; т. е. квадратный корень из –1.

Math::Trig

```
use Math::Trig;

$x = tan(0.9);
$y = acos(3.7);
$z = asin(2.4);

$halfpi = pi/2;

$rad = deg2rad(120);
```

В самом Perl определены только три тригонометрические функции: sin, cos и atan2. Модуль Math::Trig замещает их более искусными версиями и предоставляет все остальные тригонометрические функции, включая tan, csc, cosec, sec, cot, cotan, asin, acos, atan, sinh, cosh, tanh и многие другие. Кроме того, определена константа pi, а также функции преобразования радианов и градусов deg2rad и grad2rad. Предоставляется поддержка декартовых, сферических и цилиндрических систем координат. Этот модуль неявно использует Math::Complex по мере надобности (и наоборот) для вычислений, требующих комплексных чисел.

Net::hostent

```
use Socket;
use Net::hostent;
```

```
print inet_ntoa(gethost("www.perl.com")->addr);  # выводит 208.201.239.50
printf "%vd", gethost("www.perl.com")->addr;     # то же самое

print gethost("127.0.0.1")->name;                # выводит localhost

use Net::hostent ':FIELDS';
if (gethost($name_or_number)) {
    print "name is $h_name\n";
    print "aliases are $h_aliases\n";
    print "addrs are ",
        join ", " => map { inet_ntoa($_) } @h_addr_list;
}
```

Экспорт этого модуля по умолчанию замещает базовые функции gethostbyname и gethostbyaddr, заменяя их версиями, которые возвращают объект Net::hostent (или undef при неудаче). У этого объекта есть методы доступа к атрибутам, возвращающие одноименные поля из структуры библиотеки C struct hostent в *netdb.h*: name, aliases, addrtype, length и addr_list. Методы aliases и addr_list возвращают ссылки на массивы; остальные возвращают скаляры. Метод addr эквивалентен начальному элементу в массиве, на который ссылается addr_list. Функция gethost является клиентом, передающим числовой аргумент в gethostbyaddr посредством функции Socket::inet_aton, а все остальное в gethostbyname. Как и для остальных «полупрагматических» модулей, которые замещают встроенные функции, возвращающие списки, при импорте тега «:FIELDS» можно обращаться к скалярным или представляющим массивы переменным пакета под теми же именами, что и методы, добавляя ведущие «h_». При этом, однако, все равно замещаются базовые функции.

POSIX

```
use POSIX;

# Округлять действительные числа вверх или вниз до ближайшего целого.
$n = ceil($n);      # округление вверх
$n = floor($n);     # округление вниз

# Выводит "2000-04-01" для текущей даты.
$datestr = strftime("%Y-%m-%d", localtime);

# Выводит "Saturday 04/01/00" для той же даты.
$datestr = strftime("%A %D", localtime);

# Опробовать новые имена файлов, пока не получится то,
# которое еще не существует; см. также File::Temp
# в CPAN, или в v5.6.1.
do {
    $name = tmpnam();
} until sysopen(FH, $name, O_CREAT|O_EXCL|O_RDWR, 0666);
```

```
# Проверка небезопасного предоставления chown в системе.
if (sysconf(_PC_CHOWN_RESTRICTED)) {
    print "Hurray - - only the superuser may call chown\n";
}

# Найти информацию по текущему uname.
my($kernel, $hostname, $release, $version, $hardware) = uname();

use POSIX ":sys_wait_h";
while (($dead_pid = waitpid(-1, &WNOHANG)) > 0) {
    # Если хотите, можете что-то сделать с $dead_pid.
}

# Стать новым лидером сеанса/группы процессов (необходимо для создания демонов,
# на которые не действуют сигналы клавиатуры или завершение оболочки).
setsid(0)           or die "setsid failed: $!";
```

Модуль Perl `POSIX` позволяет обращаться ко всем (или почти ко всем) стандартным идентификаторам POSIX 1003.1, а также еще некоторым из ANSI C, которые мы не знали, куда поместить. В этом модуле больше функций, чем в любом другом. Подробности см. в электронной документации или в «POSIX Programmer's Guide» Дональда Льюина (Donald Lewine, O'Reilly, 1991).

Идентификаторы, которые в C представляют собой `#define` без параметров, например `EINTR` или `O_NDELAY`, автоматически экспортируются в ваше пространство имен как константные функции. Функции, обычно недоступные в Perl (например, `floor`, `ceil`, `strftime`, `uname`, `setsid`, `setlocale` и `sysconf`), по умолчанию экспортируются. Функции с такими же именами, как встроенные в Perl, например `open`, не экспортируются в отсутствие специального запроса, но, кажется, большинство программистов предпочитает полностью квалифицированные имена функций, чтобы отличить `POSIX::open` от `CORE::open`.

Несколько функций не реализовано, поскольку они специфичны для C. При попытке их вызова они выводят сообщение о том, что они не реализованы, и предлагают использовать эквивалент Perl, если таковой существует. Например, попытка обратиться к функции `setjmp` выдает сообщение `"setjmp() is C-specific: use eval {} instead"` (используйте `eval`), а `tmpfile` рекомендует `"Use method io::File::new_tmpfile()"`. (В Perl 5.6.1 следует использовать `File::Temp`.)

Модуль `POSIX` позволяет настолько приблизиться к операционной системе (или хотя бы к соответствующим частям стандартных адресов POSIX), насколько мог бы любой программист C. Это позволяет осуществлять феноменально мощные и полезные вещи, например, блокировать сигналы и управлять настройками ввода-вывода терминала. Однако это означает и то, что в итоге ваш код будет выглядеть похожим на C. Чтобы продемонстрировать, как избежать буферизации при вводе, приводим пример полной программы для получения не буферизированного посимвольного ввода в любой POSIX-системе:

```
#!/usr/bin/perl -w
use strict;
$| = 1;
for (1..4) {
    my $got;
    print "gimme: ";
    $got = getone();
    print "- -> $got\n";
}
exit;

BEGIN {
    use POSIX qw(:termios_h);
    my ($term, $oterm, $echo, $noecho, $fd_stdin);
    $fd_stdin = fileno(STDIN);
    $term     = POSIX::Termios->new();
    $term->getattr($fd_stdin);
    $oterm    = $term->getlflag();
    $echo     = ECHO | ECHOK | ICANON;
    $noecho   = $oterm & ~$echo;
    sub cbreak {
        $term->setlflag($noecho);
        $term->setcc(VTIME, 1);
        $term->setattr($fd_stdin, TCSANOW);
    }
        $term->setlflag($oterm);
        $term->setcc(VTIME, 0);
        $term->setattr($fd_stdin, TCSANOW);
    }
    sub getone {
        my $key = "";
        cbreak();
        sysread(STDIN, $key, 1);
        cooked();
        return $key;
    }
}
END { cooked() }
```

Страница руководства по модулю POSIX содержит полный список экспортируемых функций и констант. Их столько, что часто ограничиваются импортом только части из них, например ":sys_wait_h", ":sys_stat_h" или ":termios_h". Пример блокирующих сигналов для модуля POSIX приведен в главе 16.

Safe

```
use Safe;

$sandbox = Safe->new();              # анонимная песочница
$sandbox = Safe->new("PackName");    # в этой таблице символов
```

```
# Включить или выключить коды операций по группе или имени.
$sandbox->permit(qw(:base_core));
$sandbox->permit_only(qw(:base_core :base_loop :base_mem));
$sandbox->deny("die");

# как do(), но в песочнице
$ok = $sandbox->rdo($filename);

# как eval(), но в песочнице
$ok = $sandbox->reval($code);        # без 'use strict'
$ok = $sandbox->reval($code, 1);     # с 'use strict'
```

Модуль Safe пытается обеспечить ограниченную среду для защиты оставшейся части программы от опасных операций. Он применяет для этого две различные стратегии. Во многом подобно тому, как анонимный демон FTP использует *chroot*(2) для изменения вида корня файловой системы, создание объекта раздела с помощью Safe->new("PackName") изменяет взгляд этого раздела на собственное пространство имен. Теперь раздел рассматривает в качестве своей корневой таблицы символов (main::) таблицу символов, которую остальная часть программы видит как PackName::. То, что выглядит как Frobnitz:: внутри раздела, снаружи в действительности представляет собой PackName::Frobnitz::. Если не передать конструктору аргумент, то будет выбрано случайное новое имя пакета.

Вторым и более важным средством, предоставляемым разделом Safe, является заключение кода, считающегося законным, внутрь eval. Программист может подправить набор разрешенных кодов операций (допустимых операций Perl), вызывая методы своего объекта Safe. Есть два метода для компиляции кода в разделе Safe: rdo («ограниченный do») для файлов и reval («ограниченный eval») для строк. Они похожи на do с именем файла и eval со строкой, но выполняются в ограниченном пространстве имен с ограниченными кодами операций. Первый аргумент задает имя файла или строку для компиляции, а необязательный второй аргумент указывает, надо ли компилировать код в условиях use strict.

Планируется переписать этот модуль (мы хотим для большей безопасности изолировать «песочницу» в отдельном потоке интерпретатора), поэтому обратитесь к странице руководства по Safe за информацией о возможных изменениях. См. также главу 23.

Socket

```
use Socket;

$proto = getprotobyname('udp');
socket(SOCK, PF_INET, SOCK_DGRAM, $proto)
    or die "socket: $!";
$iaddr = gethostbyname('hishost.com');
$port  = getservbyname('time', 'udp');
$sin   = sockaddr_in($port, $iaddr);
```

```
send(SOCK, 0, 0, $sin)
    or die "send: $!";

$proto = getprotobyname('tcp');
socket(SOCK, PF_INET, SOCK_STREAM, $proto)
    or die "socket: $!";
$port = getservbyname('smtp', 'tcp');
$sin  = sockaddr_in($port,inet_aton("127.1"));
$sin  = sockaddr_in(7,inet_aton("localhost"));
$sin  = sockaddr_in(7,INADDR_LOOPBACK);
connect(SOCK,$sin)
    or die "connect: $!";

 ($port, $iaddr) = sockaddr_in(getpeername(SOCK));
$peer_host = gethostbyaddr($iaddr, AF_INET);
$peer_addr = inet_ntoa($iaddr);

$proto = getprotobyname('tcp');
socket(SOCK, PF_UNIX, SOCK_STREAM, $proto)
    or die "connect: $!";
unlink('/tmp/usock');   # XXX: намеренно игнорировать ошибку
$sun = sockaddr_un('/tmp/usock');
connect(SOCK,$sun)     or die "connect: $!";

use Socket qw(:DEFAULT :crlf);
# Теперь для окончания строк можно использовать CR(), LF() и CRLF()
# или $CR, $LF и $CRLF.
```

Модуль `Socket` предоставляет доступ к константам из включаемого файла библиотеки C *sys/socket.h*, чтобы использовать их в функциях Perl для работы с сокетами низкого уровня. Он предоставляет также две функции, `inet_aton` и `inet_ntoa`, для преобразования IP-адресов между ASCII (типа «127.0.0.1») и упакованным сетевым представлением, и две специальные функции упаковки-распаковки, `sockaddr_in` и `sockaddr_un`, которые обрабатывают двоичные адреса сокетов, необходимые этим вызовам низкого уровня. Тег импорта `:crlf` дает символические имена для различных соглашений по концу строки, что устраняет зависимость от интерпретации `\r` и `\n` в конкретной системе. Большинство протоколов Интернета предпочитает CRLF, но терпит LF. Стандартный модуль `io::Socket` предоставляет интерфейс более высокого уровня к TCP. См. главу 16.

Symbol

```
use Symbol "delete_package";
delete_package("Foo::Bar");
print "deleted\n" unless exists $Foo::{"Bar::"};

use Symbol "gensym";
$sym1 = getsym();       # Возвращает новый анонимный typeglob.
$sym2 = getsym();       # Еще один новый анонимный typeglob.
```

```
package Demo;
use Symbol "qualify";
$sym = qualify("x");              # "Demo::x"
$sym = qualify("x", "Foo");       # "Foo::x"
$sym = qualify("Bar::x");         # "Bar::x"
$sym = qualify("Bar::x", "Foo");  # "Bar::x"

use Symbol "qualify_to_ref";
sub pass_handle(*) {
    my $fh = qualify_to_ref(shift, caller);
    ...
}
# Теперь вы можете вызывать pass_handle с FH, "FH", *FH или \*FH.
```

Модуль Symbol предоставляет функции, облегчающие работу с глобальными именами: typeglob, именами форматов, указателями файлов, таблицами символов пакета и всем прочим, что еще может потребоваться назвать через таблицу символов. Функция delete_package полностью очищает пространство имен пакета (делая в результате безымянными любые дополнительные ссылки на объекты таблицы символов, в том числе ссылки из предварительно компилированного кода). Функция gensym при каждом вызове возвращает анонимный typeglob. (В последнее время эта функция не часто применяется, потому что появилась возможность самооживления неопределенных скаляров в правильные указатели файлов при передаче их аргументами в open, pipe, socket и прочие).

Функция qualify принимает имя, которое может не быть полностью квалифицированным, и возвращает полностью квалифицированное имя. Если ей нужно поставить впереди имя пакета, она берет его из второго аргумента (если он отсутствует, то берет имя текущего пакета). Функция qualify_to_ref работает аналогично, но создает ссылку на typeglob, представляемый символом. Это важно в функциях, которые принимают указатели файлов и каталогов или имена форматов, но которым не требуется передача их по ссылке. Например, функции, в прототипе которых задан typeglob, принимают любой из этих форматов, но не преобразуют автоматически голые слова в ссылки таблицы символов. Преобразовав такой аргумент с помощью qualify_to_ref, можно использовать переданный указатель даже при действии strict refs. Можно даже освятить его в объект, поскольку это правильная ссылка.

Sys::Hostname

```
use Sys::Hostname;
$hostname = hostname();
```

Модуль Sys::Hostname предоставляет всего одну функцию — hostname, — каковое обстоятельство компенсируется тем, что она из кожи вон лезет, пытаясь определить, как зовет себя ваш хост. В системах, поддерживающих стандартный системный вызов *gethostname*(2), применяется именно он как

самый эффективный метод.[1] В других системах используются данные, получаемые от стандартного *hostname*(1). Есть и такие системы, в которых вызывается функция *uname*(3) из библиотеки C, к которой можно также обратиться из Perl как `POSIX::uname`. Если все это не помогает, предпринимаются другие мужественные попытки. Все, что система считает разумным, Perl старается использовать. В некоторых системах имя хоста может не быть полностью квалифицировано доменным именем; если узнать его необходимо, обратитесь к модулю `Net::Domain` из CPAN.

Нужно также учитывать, что `hostname` возвращает только одно значение, однако в системе может быть несколько сетевых интерфейсов, поэтому, собираясь программировать сокеты и используя этот модуль, можно получить не то имя, которое связано с интересующим нас интерфейсом. В некоторых случаях придется, вероятно, покопаться в выводе команды *ifconfig*(8) или той, которая эквивалентна ей в данной системе.

Sys::Syslog

```
use Sys::Syslog;                          # Отсутствует setlogsock.
use Sys::Syslog qw(:DEFAULT setlogsock);  # Получает также setlogsock.

openlog($program, 'cons,pid', 'user');
syslog('info', 'this is another test');
syslog('mail|warning', 'this is a better test: %d', time());
closelog();
syslog('debug', 'this is the last test');
setlogsock('unix');
openlog("$program $$", 'ndelay', 'user');
syslog('info', 'problem was %m'); # %m == $! на языке syslog
syslog('notice', 'fooprogram: this is really done');

setlogsock("unix");             # "inet" или "unix"
openlog("myprogname", $logopt, $facility);
syslog($priority, $format, @args);
$oldmask = setlogmask($mask_priority);
closelog();
```

Модуль `Sys::Syslog` действует подобно функции *syslog*(3) библиотеки C, посылая сообщения демону системного журнала *syslogd*(8). Он особенно полезен в демонах и других программах, у которых нет терминала для получения диагностического вывода информации, а также в программах, озабоченных безопасностью и желающих сохранить более долговременные записи о своих (или чьих-нибудь еще) действиях. Поддерживаются следующие функции:

[1] Который доступен непосредственно как неэкспортированная функция `Sys::Hostname::ghname`, только никому не говорите об этом.

openlog *IDENT*, *OPTLIST*, *FACILITY*

Устанавливает соединение с дружественным демоном системного журнала. *IDENT* задает строку, которой подписываются сообщения (например, $0, имя вашей программы). *OPTLIST* является строкой опций, разделенных запятыми, например "cons", "pid" и "ndelay". *FACILITY* может быть чем-то вроде "auth", "daemon", "kern", "lpr", "mail", "news" или "user" для системных программ и одним из "local0".. "local7 для локальных. Последующие сообщения записываются с использованием заданных параметров.

syslog *PRIORITY*, *FORMAT*, *ARGS*

Посылает демону сообщение с заданным приоритетом *PRIORITY*. *FORMAT* такой же, как в printf – замещение процентов последующими аргументами, за исключением того, что по соглашениям стандартной библиотечной функции *syslog*(3) специальная последовательность "%m" вставляет в это место errno (переменная Perl $!).

setlogsock *TYPE*

TYPE должен быть "inet" или "unix". В некоторых системах демоны по умолчанию не обращают внимания на поступающие из домена Интернета системные сообщения, поэтому можно присвоить значение "unix", поскольку оно не устанавливается по умолчанию.

closelog

Обрубает соединение с демоном.

Чтобы этот модуль работал, в версиях Perl до 5.6.0 системный администратор должен был запустить *h2ph*(1) с вашим включаемым файлом *sys/syslog.h* и создать библиотечный файл *sys/syslog.ph*. Однако при установке Perl это по умолчанию не проводилось. Более поздние версии используют интерфейс XS, поэтому подготовка *sys/syslog.ph* больше не требуется.

Term::Cap

```
use Term::Cap;

$ospeed = eval {
    require POSIX;
    my $termios = POSIX::Termios->new();
    $termios->getattr;
    $termios->getospeed;
} || 9600;

$terminal = Term::Cap->Tgetent({ TERM => undef, OSPEED => $ospeed });
$terminal->Tputs('cl', 1, STDOUT);          # Очистить экран.
$terminal->Tgoto('cm', $col, $row, STDOUT); # Установить курсор.
```

Модуль Term::Cap предоставляет доступ к программам системной библиотеки *termcap*(3). Подробности см. в системной документации. Не годится для систем, где есть только *terminfo*(5) и нет *termcap*(5). (Многие системы с terminfo

умеют эмулировать termcap.) Однако в CPAN можно найти модуль Term::Info, а также Term::ReadKey, Term::ANSIColor и различные модули Curses, которые помогут осуществить ввод одной клавиши, цветную выдачу или управление терминалами на более высоком уровне, чем Term::Cap или Term::Info.

Text::Wrap

```
use Text::Wrap;  # Импортирует wrap().

@lines = (<<"EO_G&S" =~ /\S.*\S/g);

    This particularly rapid,
    unintelligible
    patter isn't generally
    heard,
    and if
    it is, it
    doesn't matter.

EO_G&S

$Text::Wrap::columns = 50;
print wrap(" " x 8, " " x 3, @lines), "\n";
```

Этот код выводит:

```
        This particularly rapid, unintelligible
   patter isn't generally heard и if it is, it
   doesn't matter.
```

Модуль Text::Wrap реализует простую программу форматирования абзацев. Его функция wrap форматирует один абзац, перенося строки по границам слов. Первый аргумент является префиксом первой возвращаемой строки. Второй – префиксом для всех строк, кроме первой. Все остальные аргументы соединяются вместе с использованием символа перевода строки как разделителя и возвращаются как одна переформатированная строка, содержащая абзац. Пользователю придется самому определить ширину своего терминала или, по крайней мере, задать то, что ему требуется, в $Text::Wrap::columns. Хотя определить число колонок можно посредством вызова ioctl TioCGWINSZ, для тех, кто не привычен к программированию на C, проще установить из CPAN модуль Term::ReadKey и использовать его программу GetTerminalSize.

Time::Local

```
use Time::Local;
$time = timelocal($sec,$min,$hours,$mday,$mon,$year);
$time = timegm($sec,$min,$hours,$mday,$mon,$year);
```

```
$time = timelocal(50, 45, 3, 18, 0, 73);
print "Scalar localtime gives: ", scalar(localtime($time)), "\n";
$time += 28 * 365.2425 * 24 * 60 * 60;
print "Twenty-eight years of seconds later, it's now\n\t",
    scalar(localtime($time)), "\n";
```

Этот код выводит:

```
Scalar localtime gives: Thu Jan 18 03:45:50 1973
Twenty-eight years of seconds later, it's now
        Wed Jan 17 22:43:26 2001
```

Модуль `Time::Local` предоставляет две функции – `timelocal` и `timegm`, – которые работают как обратные функции для стандартных функций `localtime` и `gmtime`, соответственно. Это значит, что они принимают список числовых значений для компонентов, которые `localtime` возвращает в списковом контексте, и определяют, что нужно подать на вход `localtime`, чтобы получить эти значения. Это можно сделать, если необходимо сравнить или выполнить вычисления над двумя различными датами. Хотя это не функции общего назначения для разбора даты и времени, но для тех, кто сможет вводить данные в правильном формате, их часто достаточно. Однако, как можно видеть из вышеприведенного примера, у времени есть свои особенности и даже простые вычисления часто оказываются неправильными из-за високосных лет, високосных секунд и фаз луны. Эти и другие проблемы решают два больших и полнофункциональных модуля CPAN: `Date::Calc` и `Date::Manip`.

Time::localtime

```
use Time::localtime;
printf "Year is %d\n", localtime->year() + 1900;

$now = ctime();

use Time::localtime;
use File::stat;
$date_string = ctime(stat($file)->mtime);
```

Этот модуль замещает базовую функцию `localtime`, заменяя ее версией, которая возвращает объект `Time::tm` (или `undef` при ошибке). Модуль `Time::gmtime` делает то же самое, но заменяет базовую функцию `gmtime`. Возвращаемый объект имеет методы для доступа к одноименным полям структуры `struct tm` в библиотеке C из *time.h*, а именно `sec`, `min`, `hour`, `mday`, `mon`, `year`, `wday`, `yday` и `isdst`. Функция `ctime` дает возможность обратиться к (скалярному значению) первоначальной функции `CORE::localtime`. Обратите внимание, что возвращаемые значения поступают прямо из `struct tm` и имеют такой же диапазон, как в этой структуре. Пример того, как правильно образовать год из четырех цифр, см. выше. Функция `POSIX::strftime` еще более удобна для форматирования даты и времени в различных привлекательных стилях.

User::grent

```
use User::grent;
$gr = getgrgid(0)          or die "No group zero";
if ($gr->name eq "wheel" && @{$gr->members} > 1) {
    print "gid zero name wheel, with other members";
}

$gr = getgr($whoever);  # Принимает строку или число.

use User::grent ':FIELDS';
getgrgid(0)                or die "No group zero";
if ($gr_name eq "wheel" && @gr_members > 1) {
    print "gid zero name wheel, with other members";
}
```

Экспорт этого модуля по умолчанию замещает базовые функции `getgrent`, `getgruid` и `getgrnam`, заменяя их версиями, которые возвращают объект `User::grent` (или `undef` при неудаче). У данного объекта есть методы для доступа к одноименным полям структуры `struct group` в библиотеке C из *grp.h*, а именно `name`, `passwd`, `gid` и `members` (не `mem`, как в C!). Первые три возвращают скаляры, последний – ссылку на массив. Можно также импортировать поля структуры в собственное пространство имен в качестве обычных переменных посредством тега импорта `":FIELDS"`, хотя при этом все равно замещаются базовые функции. Имена переменных (трех скаляров и массива) начинаются с `"gr_"`. Функция `getgr` представляет собой просто внешний переключатель, отправляющий числовой аргумент в `getgrgid`, а строковый – в `getgrnam`.

User::pwent

```
use User::pwent;                 # По умолчанию  замещает только встроенные.
$pw = getpwnam("daemon")   or die "No daemon user";
if ($pw->uid == 1 && $pw->dir =~ m#^/(bin|tmp)?$# ) {
    print "gid 1 on root dir";
}
$pw = getpw($whoever);           # Принимает строку или число.
$real_shell = $pw->shell || '/bin/sh';
for (($fullname, $office, $workphone, $homephone) =
       split /\s*,\s*/, $pw->gecos)
{
   s/&/ucfirst(lc($pw->name))/ge;
}

use User::pwent qw(:FIELDS);     # Устанавливает глобальные переменные в текущем
пакете.
getpwnam("daemon")         or die "No daemon user";
if ($pw_uid == 1 && $pw_dir =~ m#^/(bin|tmp)?$# ) {
    print "gid 1 on root dir";
}
```

```
use User::pwent qw/pw_has/;
if (pw_has(qw[gecos expire quota])) { .... }
if (pw_has("name uid gid passwd"))  { .... }
printf "Your struct pwd supports [%s]\n", scalar pw_has();
```

Экспорт этого модуля по умолчанию замещает базовые функции getpwent, getpwuid и getpwnam, заменяя их версиями, которые возвращают объект User::pwent (или undef при неудаче). Часто лучше использовать этот модуль, а не замещаемые им функции, поскольку встроенные функции во имя достижения обратной совместимости перегружают или даже пропускают различные позиции в возвращаемом списке.

У возвращаемого объекта есть методы для доступа к аналогично названным полям структуры C passwd из *pwd.h*, лишенные ведущей части "pw_", а именно name, passwd, uid, gid, change, age, quota, comment, class, gecos, dir, shell и expire. Поля passwd, gecos и shell являются мечеными. Можно также импортировать поля структуры в собственное пространство имен в качестве обычных переменных посредством тега импорта ":FIELDS", хотя при этом все равно замещаются базовые функции. Обращайтесь к этим полям как к скалярным переменным, добавляя "pw_" перед именем метода. Функция getpw является просто внешним переключателем, который отправляет числовой аргумент в getpwuid, а строковый аргумент в getpwnam.

Perl считает, что не бывает машин, в которых реализовано более одного из методов change, age или quota, а также более одного из comment или class. Некоторые машины не поддерживают expire, gecos и, возможно, даже passwd. Можно вызывать эти методы на любой машине, но если они не реализованы, то возвращают undef. Подробности см. в *passwd*(5) и *getpwent*(3).

Можно определить, реализованы ли эти поля, запросив для них импортируемую функцию pw_has. Она возвращает значение «истина», если все параметры являются поддерживаемыми полями на платформе компиляции, и значение «ложь», если одно или более полей не реализовано, а также возбуждает исключительную ситуацию, если получает запрос о поле, имени которого она не знает. Если вызвать ее без аргументов, она возвращает список полей, которые библиотека C пользователя полагает поддерживающимися.

Интерпретация поля gecos может зависеть от системы, но часто включает четыре поля, разделенные запятыми и содержащие полное имя пользователя, адрес офиса, рабочий телефон и домашний телефон. Символ & в поле gecos должен быть заменен именем регистрации пользователя с учетом букв верхнего регистра. Поле shell, если оно пусто, должно полагаться равным */bin/sh*, хотя Perl не делает этого за вас. Поле passwd – это хешированная в одну сторону неразбериха, а не открытый текст, и раскрыть его исходное значение, кроме как грубым перебором, нельзя. В защищенных системах часто применяется более надежное хеширование, чем DES. В системах, поддерживающих теневые пароли, Perl автоматически возвращает запись теневого пароля, если пользователь имеет для этого достаточные права, и даже если поставленная с системой библиотека C слишком близорука, чтобы понять, что она должна это сделать.

33

Диагностические сообщения

Эти сообщения классифицируются следующим образом (перечислены по возрастанию степени вызываемого отчаяния):

Класс	Значение
(W)	Warning – предупреждение (необязательная)
(D)	Deprecation – протест (необязательная)
(S)	Severe warning – серьезное предупреждение (обязательная)
(F)	Fatal error – фатальная ошибка (перехватываемая)
(P)	Panic – внутренняя ошибка, которую вы никогда не должны видеть (перехватываемая)
(X)	Абсолютно фатальная ошибка (не перехватываемая)
(A)	Alien – сообщение о чужой ошибке (генерируемой не Perl)

Большинством сообщений первых трех классов (W, D и S) можно управлять с помощью прагмы `warnings` или ключей `-w` и `-W`. Если сообщением можно управлять с помощью директивы `warnings`, его категория указывается после классифицирующей буквы; например (W misc) указывает на предупреждение по разным поводам. Директива `warnings` описана в главе 31 «Модули прагм».

Предупреждения можно перехватывать, а не выводить, если присвоить `$SIG{__WARN__}` ссылку на программу, которая будет вызываться для каждого предупреждения. Можно также перехватить управление до того, как пере-

хватываемая ошибка «умрет», если присвоить `$SIG{__DIE__}` ссылку на подпрограмму, но если не вызвать `die` из обработчика, то при возврате из него все равно будет возбуждена исключительная ситуация. Иными словами, нельзя таким способом «дефатализировать» ошибку. Для этого нужно выполнить `eval`.

Предупреждения по умолчанию всегда включены, за исключением тех случаев, когда они явно выключены прагмой `warnings` или ключом `-X`.

В следующих сообщениях *%s* означает интерполируемую строку, которая определена только при генерации сообщения. (Аналогично *%d* означает интерполируемое число – можете вспомнить форматы `printf`, но здесь мы посредством *%d* обозначаем числа по любому основанию.) Обратите внимание, что некоторые сообщения *начинаются* с *%s*, что делает проблематичным перечисление их в алфавитном порядке. Если найти в предполагаемом месте то, что вам нужно, не удается, то следует искать среди таких сообщений. Символы `"%-?@` при сортировке идут впереди алфавитных символов, а `[` и `\` после них.

Решив, что возникшая ошибка является ошибкой Perl, а не вашей, следует свести ее к минимальному контрольному примеру и сообщить о ней с помощью программы *perlbug*, поставляемой с Perl.

`"%s" variable %s masks earlier declaration in same %s`

(W misc) Переменная `my` или `our` повторно объявлена в текущей области видимости или команде, что маскирует все обращения к предыдущему экземпляру. Это почти типографская ошибка. Обратите внимание, что прежняя переменная продолжает существовать до конца области видимости или пока в замыкании не будут разрушены все объекты ссылки на нее.

`"my sub" not yet implemented`

(F) Подпрограммы с лексической областью видимости еще не реализованы. Пока не пытайтесь использовать их.

`"my" variable %s can't be in a package`

(F) Переменные с лексической областью видимости размещаются не в пакете, поэтому не имеет смысла пытаться объявлять их с квалификатором пакета. Укажите спецификатор `local`, если необходимо локализовать переменную пакета.

`"no" not allowed in expression`

(F) Ключевое слово `no` распознается и выполняется на этапе компиляции, не возвращая никакого полезного значения.

`"our" variable %s redeclared`

(W misc) Видимо, однажды эта глобальная переменная уже была объявлена в текущей лексической области видимости.

`"use" not allowed in expression`

(F) Ключевое слово `use` распознается и выполняется на этапе компиляции и не возвращает никакого значения.

`'!' allowed only after types %s`

(F) Символ `'!'` разрешается в `pack` и `unpack` только после определенных типов.

`'|' and '<' may not both be specified on command line`

(F) Это специфическая для VMS ошибка. Perl осуществляет собственную переадресацию в командной строке. Он обнаружил, что `STDIN` является каналом, и вы также попытались перенаправить `STDIN` с помощью `<`. У клиента может быть только один `STDIN`.

`'|' and '>' may not both be specified on command line`

(F) Это специфическая для VMS ошибка. Perl осуществляет собственную переадресацию в командной строке и полагает, что пользователь пытается переадресовать `STDOUT` одновременно в файл и канал к другой программе. Нужно выбрать одно или другое, хотя ничто не мешает открыть канал в программу или сценарий Perl, который расщепляет вывод на два потока, например:

```
open(OUT,">$ARGV[0]") or die "Can't write to $ARGV[0]: $!";
while (<STDIN>) {
    print;
    print OUT;
}
close OUT;
```

`/ cannot take a count`

(F) Шаблон `unpack` задавал вычисление длины строки, но был указан и явный размер строки.

`/ must be followed by a, A, or Z`

(F) Шаблон `unpack` задавал вычисление длины строки, за которым должна следовать одна из букв `a`, `A` или `Z` для указания того, какого типа строку нужно распаковать.

`/ must be followed by a*, A*, or Z*`

(F) Шаблон `pack` задавал вычисление длины строки. В настоящее время вычисление длины можно произвести только для `a*`, `A*` или `Z*`.

`/ must follow a numeric type`

(F) Шаблон `unpack` содержал `#`, но не соответствовал каким-то числовым спецификациям `unpack`.

`% may only be used in unpack`

(F) Нельзя упаковывать строку, передавая контрольную сумму, потому что в процессе контрольного суммирования теряется информация, и обратный процесс будет невозможен.

`Repeat count in pack overflows`

(F) Нельзя задавать счетчик повторения `pack` таким большим, чтобы он превосходил целое со знаком.

`Repeat count in unpack overflows`

(F) Нельзя задавать счетчик повторения `unpack` таким большим, чтобы он превосходил целое со знаком.

`/ %s/: Unrecognized escape \\%c passed through`

(W regexp) Задана комбинация с символом обратного слэша, неизвестная Perl. Эта комбинация находится в интерполируемой переменной или регулярном выражении, ограниченном одинарными кавычками `'`. Символ был воспринят буквально.

`/ %s/: Unrecognized escape \\%c in character class passed through`

(W regexp) В классе символов указана комбинация с символом обратного слэша, неизвестная Perl. Символ был воспринят буквально.

`/ %s/ should probably be written as "%s"`

(W syntax) Использован шаблон, в котором Perl рассчитывал найти строку, как в первом аргументе `join`. Perl будет рассматривать истинный или ложный результат поиска шаблона в `$_` как строку, что, вероятно, не входило в ваши намерения.

`%s (...) interpreted as function`

(W syntax) Вы столкнулись с правилом, гласящим, что списковый оператор, за которым идут круглые скобки, превращается в функцию с аргументами спискового оператора, заключенными в скобки.

`%s () called too early to check prototype`

(W prototype) Обращение к функции, имеющей прототип, произошло раньше, чем анализатор обнаружил ее определение или объявление, и Perl не смог проверить соответствие вызова прототипу. Следует либо добавить в начало объявление прототипа для рассматриваемой подпрограммы, либо переместить определение подпрограммы вперед ее вызова, чтобы обеспечить правильную проверку прототипа. Можно также, если есть уверенность в правильном вызове функции, поместить амперсанд перед ее именем, чтобы избежать предупреждения.

`%s argument is not a HASH or ARRAY element`

(F) Аргумент `exists` должен быт элементом хеша или массива, например:

```
$foo{$bar}
$ref->{"susie"}[12]
```

`%s argument is not a HASH or ARRAY element or slice`

(F) Аргумент `delete` должен быт элементом хеша или массива, например:

```
$foo{$bar}
$ref->{"susie"}[12]
```

или срезом хеша или массива, например:

```
@foo[$bar, $baz, $xyzzy]
@{$ref->[12]}{"susie", "queue"}
```

`%s argument is not a subroutine name`

(A) Аргумент `exists` в `exists &sub` должен быть именем подпрограммы, а не вызовом ее. Такую ошибку вызывает `exists &sub()`.

`%s did not return a true value`

(F) При загрузке файла с применением `require` (или `use`) должно возвращаться истинное значение, указывающее на его правильную компиляцию и выполнение кода инициализации. Принято ставить в конце такого файла `1;`, хотя это может быть любое истинное значение.

`%s found where operator expected`

(S) Лексический анализатор Perl знает, что должно следовать – терм или оператор. Если он видит то, что считает термом, когда рассчитывал увидеть оператор, то выдает это предупреждение. Обычно оно свидетельствует о том, что был пропущен оператор или разделитель, например точка с запятой.

`%s had compilation errors`

(F) Последнее итоговое сообщение при неудаче *perl -c*.

`%s has too many errors`

(F) Анализатор отказался от анализа программы после обнаружения 10 ошибок. Дальнейшие сообщения об ошибках могут оказаться бессодержательными.

`%s matches null string many times`

(W regexp) Заданный шаблон создал бы бесконечный цикл, если бы механизм регулярных выражений этого не проверил.

`%s never introduced`

(S internal) Указанный символ был объявлен, но вышел из области видимости до того, как был использован.

`%s package attribute may clash with future reserved word: %s`

(W reserved) Было задано имя атрибута из строчных букв, имеющего обработчик в пакете. Это имя, возможно, когда-нибудь будет иметь для Perl какой-то смысл, хотя пока это не так. Возможно, следует использовать в имени атрибута буквы разных регистров.

`%s syntax OK`

(F) Последнее итоговое сообщение при успехе *perl -c*.

`%s: Command not found`

(A) Сценарий случайно запущен через *csh* вместо Perl. Проверьте строку `#!` или вручную передайте свой сценарий Perl с помощью *perl scriptname*.

`%s: Expression syntax`

(A) Сценарий случайно запущен через *csh* вместо Perl. Проверьте строку `#!` или вручную передайте свой сценарий Perl с помощью *perl scriptname*.

`%s: Undefined variable`

(A) Сценарий запущен через *csh* вместо Perl. Проверьте строку `#!` или вручную передайте свой сценарий Perl с помощью *perl scriptname*.

`%s: not found`

(A) Сценарий запущен через *csh* вместо Perl. Проверьте строку `#!` или вручную передайте свой сценарий Perl с помощью *perl scriptname*.

`(in cleanup) %s`

(W misc) Этот префикс обычно указывает на то, что данную исключительную ситуацию возбудил метод `DESTROY`. Поскольку деструкторы обычно вызываются системой в произвольные моменты выполнения и часто большое число раз, предупреждение выводится только один раз для любого количества ошибок, иначе повторялось бы одно и то же сообщение.

Это предупреждение может также явиться результатом обратных вызовов пользователя, отправляемых с флагом `G_KEEPERR`. См. *perlcall*(1).

`(Missing semicolon on previous line?)`

(S) Это предположение, которое возникает вместе с сообщением `"%s found where operator expected"`. Не следует автоматически ставить точку с запятой в предыдущей строке, увидев это сообщение.

`-P not allowed for setuid/setgid script`

(F) Сценарий должен быть открыт препроцессором C по имени, что ведет к ситуации гонки, нарушающей безопасность.

`-T and -B not implemented on filehandles`

(F) Perl не может читать из буфера стандартного ввода-вывода указателей файлов, когда не знает, какой именно это стандартный ввод-вывод. Укажите вместо этого имя файла.

`-p destination: %s`

(F) Произошла ошибка при неявном выводе, вызванном ключом командной строки `-p`. (Этот вывод идет на `STDOUT`, если он не переадресован с помощью `select`.)

`500 Server error`

См. `Server error`.

`?+* follows nothing in regexp`

(F) Регулярное выражение начато с квантификатора. Поставьте обратную косую черту, если вам нужен буквальный символ.

`@ outside of string`

(F) Шаблон упаковки задавал абсолютное положение за пределами распаковываемой строки.

`<> should be quotes`

(F) Было написано `require <file>` вместо необходимого `require 'file'`.

`\1 better written as $1`

(W syntax) Вне шаблонов ссылки на найденный текст действуют как переменные. Обратный слэш допускается в правой части замены, но стилистически лучше использовать форму переменной, поскольку этого будут ожидать другие программисты Perl, и это лучше работает, когда ссылок больше девяти.

`accept() on closed socket %s`

(W closed) Попытка выполнить `accept` на закрытом сокете. Вы не забыли проверить значение, возвращенное вызовом `socket`?

`Allocation too large: %lx`

(X) Нельзя выделять более 64 Кбайт памяти на машине MS-DOS.

`Applying %s to %s will act on scalar(%s)`

(W misc) Операторы поиска по шаблону (`//`), подстановки (`s///`) и транслитерации (`tr///`) работают со скалярными величинами. Если применить один из них к массиву или хешу, то массив или хеш будут преобразованы в скаляры – длину массива или число элементов хеша, а затем будут выполняться с этим скаляром, что, вероятно, не было вашей целью.

`Arg too short for msgsnd`

(F) `msgsnd` требует строки длиной не менее `sizeof(long)`.

`Ambiguous use of %s resolved as %s`

(W ambiguous | S) Было сказано нечто, что может быть интерпретировано не так, как предполагалось. Обычно довольно легко устранить двусмысленность с помощью кавычек, оператора, круглых скобок или объявления.

`Ambiguous call resolved as CORE:: %s (), qualify as such or use &`

(W ambiguous) Объявлена подпрограмма, имя которой совпадает с ключевым словом Perl, и использовано неквалифицированное имя для вызова одного или другого. Perl решил вызвать встроенный символ, потому что подпрограмма не импортирована.

Для ее принудительной интерпретации как вызова подпрограммы поместите перед именем амперсанд или квалифицируйте его именем пакета. Другой возможностью является импорт подпрограммы (или видимость импорта с помощью директивы `use subs`).

Чтобы интерпретировать это как оператор Perl и предотвратить вывод предупреждений, укажите префикс `CORE::` (например, `CORE::log($x)`) или объявите подпрограмму как метод объекта.

`Args must match #! line`

(F) Эмулятор setuid требует, чтобы аргументы, с которыми был вызван Perl, соответствовали указанным в строке `#!`. В некоторых системах в строке `#!` может быть всего один аргумент, поэтому попробуйте объединить несколько ключей, например, замените `-w -U` на `-wU`.

`Argument "%s" isn't numeric`

(W numeric) Указанная строка была передана в качестве аргумента оператору, который ожидал числового значения. Если вам повезет, то сообщение укажет оператор, которому так не повезло.

`Array @%s missing the @ in argument %d of %s ()`

(D deprecated) Очень старые версии Perl позволяют опустить `@` в имени массива в некоторых местах. Сейчас это сурово осуждается.

`assertion botched: %s`

(P) Внутренняя ошибка в пакете `malloc`, поставляемом с Perl.

`Assertion failed: file "%s"`

(P) Общее утверждение не выполнено. Следует изучить указанный файл.

`Assignment to both a list and a scalar`

(F) В случае присваивания условному оператору второй и третий аргументы должны оба быть либо скалярами, либо списками. В противном случае Perl не будет знать, какой контекст предоставить правой части.

`Attempt to free non-arena SV: 0x%lx`

(P internal) Все объекты SV должны размещаться из областей, в которых при выходе осуществляется уборка мусора. Был обнаружен SV, находящийся вне таких областей.

`Attempt to free nonexistent shared string`

(P internal) Perl поддерживает внутреннюю таблицу строк со счетчиками ссылок для оптимизации хранения и доступа к ключам хешей и другим строкам. Это сообщение указывает на попытку уменьшить счетчик ссылок для строки, которой нет в таблице.

`Attempt to free temp prematurely`

(W debugging) Умершие значения должны освобождаться с помощью внутренней программы `free_tmps`. Это сообщение указывает, что кто-то еще пытается освободить SV до программы `free_tmps`, из-за чего `free_tmps` будет, возможно, впоследствии пытаться освободить скаляр, на который нет ссылки.

`Attempt to free unreferenced glob pointers`

(P internal) Счетчик ссылок «сломался» на псевдонимах символов.

`Attempt to free unreferenced scalar`

(W internal) Perl собрался уменьшить счетчик ссылок скаляра и, проверяя, не станет ли он нулем, обнаружил, что он уже равен 0 и должен быть освобожден, а возможно, уже освобожден. Это может указывать на то, что `SvREFCNT_dec` вызывалась избыточное количество раз или `SvREFCNT_inc` вызывалась недостаточное количество раз, или что SV была умерщвлена, хотя этого не должно было быть, или произошла порча памяти.

`Attempt to join self`

(F) Попытка присоединения к потоку изнутри его самого, что является невыполнимой задачей. Или был указан неверный поток для `join`, или необходимо переместить `join` в другой поток.

`Attempt to pack pointer to temporary value`

(W pack) Попытка передачи временного значения (такого как результат функции или вычисления выражения) в шаблон `p` функции `pack`. Это означает, что результат содержит указатель на адрес, который в любой момент может стать недействительным, даже до окончания выполнения текущей команды. Используйте литералы или глобальные значения в качестве аргументов шаблона `p` в `pack`, чтобы избежать этого предупреждения.

`Attempt to use reference as lvalue in substr`

(W substr) Ссылка передана в качестве первого аргумента `substr`, используемой как левое значение, что несколько странно. Возможно, вы забыли ее разыменовать.

`Bad arg length for %s, is %d, should be %d`

(F) Передан буфер неправильного размера в одну из `msgctl`, `semctl` или `shmctl`. В языке C правильными размерами являются, соответственно, `sizeof(struct msqid_ds *)`, `sizeof(struct semid_ds *)` и `sizeof(struct shmid_ds *)`.

`Bad filehandle: %s`

(F) Символ был передан туда, где ожидался указатель файла, но с этим символом не связан никакой указатель файла. Возможно, `open` не выполнен или это сделано в другом пакете.

`Bad free() ignored`

(S malloc) Внутренняя программа вызвала `free` для того, чему ранее не выделялась память через `malloc`. Предупреждение обязательное, но может быть отключено установкой переменной окружения `PERL_BADFREE`, равной 1.

Это предупреждение довольно часто встречается с `DB_File` в системах с «жесткой» динамической компоновкой, например AIX и OS/2. Является программной ошибкой в Berkeley DB.

`Bad hash`

(P) Одной из внутренних программ хеширования был передан нулевой указатель HV.

`Bad index while coercing array into hash`

(F) Индекс, отыскиваемый в хеше и найденный как нулевой элемент псевдохеша, недопустим. Значения индекса должны быть 1 и выше.

`Bad name after %s::`

(F) Имя символа начато с префикса пакета и не закончено. В частности нельзя интерполировать вне кавычек, поэтому:

```
$var = 'myvar';
$sym = mypack::$var;
```

не то же самое, что:

```
$var = 'myvar';
$sym = "mypack::$var";
```

`Bad realloc() ignored`

(S malloc) Внутренняя программа вызвала `realloc` для того, чему ранее не выделялась память через `malloc`. Обязательное предупреждение, но может быть отключено установкой переменной окружения `PERL_BADFREE`, равной 1.

`Bad symbol for array`

(P) Внутренний запрос попросил добавить элемент массива к чему-то, не являющемуся элементом таблицы символов.

`Bad symbol for filehandle`

(P) Внутренний запрос попросил добавить элемент указателя файла к чему-то, не являющемуся элементом таблицы символов.

`Bad symbol for hash`

(P) Внутренний запрос попросил добавить элемент хеша к чему-то, не являющемуся элементом таблицы символов.

`Badly placed ()'s`

(A) Сценарий случайно запущен через *csh* вместо Perl. Проверьте строку `#!` или вручную передайте свой сценарий Perl с помощью *perl scriptname*.

`Bareword "%s" not allowed while "strict subs" in use`

(F) При действии прагмы `strict subs` голые слова разрешены только как идентификаторы подпрограмм, в фигурных скобках или слева от символа `=>`. Может быть, следовало предпочесть объявление подпрограммы?

`Bareword "%s" refers to nonexistent package`

(W bareword) Использовано квалифицированное голое слово вида `Foo::`, но компилятор еще не встречал такого пространства имен. Может быть, надо объявить пакет?

`Bareword found in conditional`

(W bareword) Компилятор обнаружил голое слово там, где он ожидал условный оператор, что часто указывает на разбор `||` или `&&` как части последнего аргумента предшествующей конструкции, например:

```
open FOO || die;
```

Часто это указывает на константу, введенную с ошибкой, которая интерпретирована как голое слово:

```
use constant TYPO => 1;
if (TYOP) { print "foo" }
```

Чтобы избежать таких ошибок, неплохо использовать прагму `strict`.

`BEGIN failed--compilation aborted`

(F) Не перехваченная исключительная ситуация во время выполнения подпрограммы `BEGIN`. Компиляция немедленно прекращается, а работа интерпретатора завершается.

`BEGIN not safe after errors- -compilation aborted`

(F) Perl обнаружил подпрограмму `BEGIN` (или директиву `use`, которая влечет `BEGIN`) после того, как уже произошло несколько ошибок компиляции. Поскольку нельзя гарантировать для `BEGIN` надлежащего окружения (из-за ошибок), а последующий код, вероятно, зависит от ее правильной работы, Perl прекращает компиляцию.

`Binary number > 0b11111111111111111111111111111111 non-portable`

(W portable) Заданное двоичное число больше `2**32-1` (4 294 967 295) и потому не переносимо между системами.

`bind() on closed socket %s`

(W closed) Попытка выполнения `bind` на закрытом сокете. А вы не забыли проверить значение, возвращенное вызовом `socket`?

`Bit vector size > 32 non-portable`

(W portable) Использование битовых векторов длиннее 32 не переносимо.

`Bizarre copy of %s in %s`

(P) Perl обнаружил попытку копирования внутреннего значения, которое копировать нельзя.

`Buffer overflow in prime_env_iter: %s`

(W internal) Специфическое для VMS предупреждение. Готовясь обойти `%ENV`, Perl обнаружил слишком длинное логическое имя или определение символа, поэтому оно было укорочено до указанной строки.

`Callback called exit`

(F) Подпрограмма, вызванная из внешнего пакета через `call_sv`, завершила работу вызовом `exit`.

`Can't "goto" out of a pseudo block`

(F) Команда `goto` была выполнена для выхода из того, что похоже на блок, но блоком, в сущности, не является. Обычно это происходит при попытке выскочить из блока сортировки или подпрограммы, чего делать «ни-ни».

`Can't "goto" into the middle of a foreach loop`

(F) Команда `goto` была выполнена для входа в середину цикла `foreach`. Попасть туда нельзя.

`Can't "last" outside a loop block`

(F) Команда `last` была выполнена для выхода из текущего блока, но это все та же проблема, когда текущий блок не существует. Обратите внимание, что блоки `if` или `else` не считаются «циклическими» блоками, так же как и блоки `sort`, `map` или `grep`. Однако обычно можно удвоить фигур-

ные скобки, поскольку внутренние фигурные скобки будут считаться циклом, выполняемым один раз.

`Can't "next" outside a loop block`

(F) Команда `next` была выполнена для повторения текущего блока, но текущий блок не существует. Обратите внимание, что блоки `if` или `else` не считаются «циклическими» блоками, так же как и блоки `sort`, `map` или `grep`. Однако обычно можно удвоить фигурные скобки, поскольку внутренние фигурные скобки будут считаться циклом, выполняемым один раз.

`Can't read CRTL environ`

(S) Специфическое для VMS предупреждение. Perl попытался прочесть элемент `%ENV` из внутреннего массива окружения CRTL и обнаружил, что массив остутствует. Необходимо выяснить, куда CRTL переместила свой массив окружения, или определить `PERL_ENV_TABLES` (см. *perlvms*(1)) так, чтобы исключить поиск этого массива.

`Can't "redo" outside a loop block`

(F) Команда `redo` была выполнена для перезапуска из текущего блока, но текущий блок не существует. Обратите внимание, что блоки `if` или `else` не считаются «циклическими» блоками, так же как и блоки `sort`, `map` или `grep`. Однако обычно можно удвоить фигурные скобки, поскольку внутренние фигурные скобки будут считаться циклом, выполняемым один раз.

`Can't bless non-reference value`

(F) Освящать можно только жесткие ссылки. Таким способом Perl осуществляет инкапсуляцию объектов.

`Can't break at that line`

(S internal) Это предупреждение предназначено для вывода только при выполнении через отладчик и указывает, что заданный номер строки не содержит команды, на которой можно остановиться.

`Can't call method "%s" in empty package "%s"`

(F) Вы правильно вызвали метод, который правильно указал на пакет, выступающий классом, но в этом пакете ничего не определено, не только методы.

`Can't call method "%s" on unblessed reference`

(F) Вызов метода должен знать, в каком пакете предполагается его выполнение. Обычно он определяет это из передаваемой ссылки на объект, но в данном случае не передана ссылка на объект. Ссылка не есть ссылка на объект, пока она не освящена.

`Can't call method "%s" without a package or object reference`

(F) Использован синтаксис вызова метода, но позиция для ссылки на объект или для имени пакета содержит выражение, возвращающее определенное значение, которое не является ни ссылкой на объект, ни именем пакета. Такую ошибку может создавать следующий код:

```
$BADREF = 42;
process $BADREF 1,2,3;
$BADREF->process(1,2,3);
```

`Can't call method "%s" on an undefined value`

(F) Использован синтаксис вызова метода, но позиция для ссылки на объект или для имени пакета содержит неопределенное выражение. Такую ошибку может создавать следующий код:

```
$BADREF = undef;
process $BADREF 1,2,3;
$BADREF->process(1,2,3);
```

`Can't chdir to %s`

(A) Мы вызвали *perl -x/foo/bar*, но */foo/bar* не является каталогом, в который можно перейти по `chdir`, возможно, потому, что его не существует.

`Can't check filesystem of script "%s" for nosuid`

(P) По какой-то причине невозможно проверить `nosuid` в файловой системе сценария.

`Can't coerce %s to integer in %s`

(F) Некоторые типы SV, в частности записи таблицы символов (typeglob), нельзя заставить быть чем-то иным. Поэтому нельзя говорить такие вещи:

```
*foo += 1;
```

Можно говорить:

```
$foo = *foo;
$foo += 1;
```

но тогда `$foo` больше не содержит glob.

`Can't coerce %s to number in %s`

(F) Некоторые типы SV, в частности записи таблицы символов (typeglob), нельзя заставить быть чем-то иным.

`Can't coerce %s to string in %s`

(F) Некоторые типы SV, в частности записи таблицы символов (typeglob), нельзя заставить быть чем-то иным.

`Can't coerce array into hash`

(F) Массив использован там, где ожидался хеш, но в массиве нет данных о том, как отображать ключи в индексы массива. Это можно делать только с массивами, в которых под индексом 0 расположена ссылка на хеш.

`Can't create pipe mailbox`

(P) Специфическая для VMS ошибка. Процесс страдает от того, что исчерпаны квоты, или из-за других проблем «канализации».

`Can't declare class for non-scalar %s in "%s"`

(S) В настоящее время только скалярные переменные можно определять, квалифицируя конкретным классом, в объявлениях `my` или `our`. В будущем семантика может быть расширена на другие типы переменных.

`Can't declare %s in "%s"`

(F) Только скаляры, массивы и хеши могут объявляться как переменные `my` или `our`. В качестве имен они должны иметь обычные идентификаторы.

`Can't do inplace edit on %s: %s`

(S inplace) Создание нового файла оказалось невозможным по указанной причине.

`Can't do inplace edit without backup`

(F) Вы работаете в системе типа MS-DOS, которая сбивается при попытке чтения из удаленного (но все еще открытого) файла. Необходимо сказать *-i.bak* или что-то похожее.

`Can't do inplace edit: %s would not be unique`

(S inplace) Файловая система не поддерживает имена длиннее 14 символов, и Perl не смог создать уникальное имя файла при редактировании по месту с ключом `-i`. Файл был проигнорирован.

`Can't do inplace edit: %s is not a regular file`

(S inplace) Попытка использования ключа `-i` со специальным файлом, таким как файл в */dev*, или FIFO. Файл был проигнорирован.

`Can't do setegid!`

(P) Вызов `setegid` при эмуляции setuid в *suidperl* по какой-то причине оказался неуспешным.

`Can't do seteuid!`

(P) Эмулятор setuid *suidperl* по какой-то причине не смог выполниться.

`Can't do setuid`

(F) Как правило, это означает, что обычный *perl* попытался выполнить `exec` *suidperl* для эмуляции setuid, но не смог. При этом выполняется поиск имени формата *sperl5.000* в том же каталоге, где находится исполняемый модуль Perl с именем *perl5.000*, обычно в */usr/local/bin* на машинах Unix. Если файл там есть, проверьте право доступа на исполнение. Если его нет, узнайте у системного администратора, почему.

`Can't do waitpid with flags`

(F) На этой машине нет `waitpid` и `wait4`, поэтому эмулируется только `waitpid` без флагов.

`Can't do {n,m} with n > m`

(F) Минимумы должны быть меньше или равны максимумам. Если необходимо, чтобы регулярное выражение находило что-то 0 раз, используйте `{0}`.

`Can't emulate -%s on #! line`

(F) В строке `#!` задан ключ, не имеющий смысла в данном месте. Например, довольно глупо помещать `-x` в строку `#!`.

`Can't exec "%s": %s`

(W exec) Вызов `system`, `exec` или канализованного `open` не смог выполнить данную программу по указанной причине. Типичными причинами могут быть неверные права доступа к файлу, отсутствие файла в `$ENV{PATH}`, компиляция данного исполняемого модуля для другой архитектуры или задание в строке сценария `#!` интерпретатора, который не может быть выполнен по аналогичным причинам. (А может быть, система вообще не поддерживает `#!`.)

`Can't exec %s`

(F) Perl пытался выполнить указанную программу, потому что это было сказано в строке `#!`. Если это не то, чего вы хотели, то упомяните `perl` в каком-нибудь месте строки `#!`.

`Can't execute %s`

(F) Был указан ключ `-S`, но у экземпляров сценария для выполнения, найденных в `PATH`, не оказалось необходимых прав доступа.

`Can't find %s on PATH, '.' not in PATH`

(F) Был указан ключ `-S`, но экземпляры сценария для выполнения не были найдены в `PATH` или у них не оказалось необходимых прав доступа. Сценарий есть в текущем каталоге, но `PATH` запрещает его использование.

`Can't find %s on PATH`

(F) Ключ `-S` был задан, но сценарий для выполнения не найден в `PATH`.

`Can't find label %s`

(F) Оператор `goto` с меткой, которой нет там, куда можно перейти.

`Can't find string terminator %s anywhere before EOF`

(F) Строки (strings) Perl могут занимать несколько строчек (lines). Это сообщение означает, что пропущен последний ограничитель. Поскольку операторы кавычек учитывают уровни вложенности, то в следующей команде отсутствует завершающая круглая скобка:

```
print q(The character '(' starts a side comment.);
```

Если это сообщение получено из встроенного документа, то возможно, что присутствует незаметный пробельный символ до или после замыкающего тега. Хороший текстовый редактор должен помогать находить такие символы.

`Can't fork`

(F) Фатальная ошибка при попытке выполнить `fork`.

`Can't get filespec - stale stat buffer?`

(S) Это предупреждение специфично для VMS. Оно происходит из-за различий в проверке доступа в VMS и в модели Unix, которая принимается Perl. В VMS проверка доступа происходит по имени файла, а не по битам в буфере `stat`, поэтому можно учитывать ACL и другие способы защиты. К несчастью, Perl полагает, что в буфере `stat` есть вся необходимая информация, и передает его вместо спецификации файла в программу проверки доступа. Она будет пытаться извлечь спецификацию файла, используя имя устройства и FID из буфера `stat`, но если выполнялись новые вызовы программы CRTL `stat`, это будет работать неправильно, поскольку каждый новый вызов переписывает имя устройства. Если появилось это предупреждение, значит, поиск имени закончился неудачей, процедура проверки доступа была прервана и возвратила ложное значение, проявив осторожность. (Замечание: программе проверки доступа известно об операторе Perl `stat` и проверках файла, поэтому данное предупреждение никогда не появится в ответ на команду Perl; оно возникает, только если какой-нибудь внутренний код легкомысленно относится к буферам `stat`.)

`Can't get pipe mailbox device name`

(P) Эта ошибка специфична для VMS. После создания почтового ящика, действующего как канал, Perl не может получить его имя для использования в дальнейшем.

`Can't get SYSGEN parameter value for MAXBUF`

(P) Эта ошибка специфична для VMS. Perl поинтересовался в `$GETSYI`, какого размера должны быть буферы почтовых ящиков, и не получил ответа.

`Can't goto subroutine outside a subroutine`

(F) Этот «очень магический» вызов `goto` *`SUBROUTINE`* может только заменить вызов одной подпрограммы на вызов другой. Он не может придумать его. Во всяком случае этот вызов следует выполнять только из подпрограммы `AUTOLOAD`.

`Can't goto subroutine from an eval-string`

(F) Вызов `goto` *`SUBROUTINE`* нельзя использовать для выхода из строки, выполняемой в `eval`. (С его помощью можно организовать выход из `eval` *`BLOCK`*, но этого, вероятно, не требуется.)

`Can't ignore signal CHLD, forcing to default`

(W signal) Perl обнаружил, что выполняется с отключенным сигналом `SIGCHLD` (иногда известным как `SIGCLD`). Поскольку отключение этого сигнала помешает правильному определению статуса завершения дочерних процессов, Perl восстановил сигнал в значение по умолчанию. Это обычно указывает на то, что родительская программа, которая могла запустить Perl (например, *cron*), очень небрежна.

`Can't localize through a reference`

(F) Было сказано что-то типа `local $$ref`, что Perl в настоящее время не может обрабатывать, поскольку, когда, покончив с областью видимости `local`, он соберется восстанавливать прежнее значение того, на что указывала `$ref`, он не сможет быть уверен, что `$ref` все еще является ссылкой.

`Can't localize lexical variable %s`

(F) Спецификатор `local` указан для переменной, которая ранее была объявлена как лексическая с помощью `my`. Это недопустимо. Для того чтобы локализовать переменную пакета с тем же именем, квалифицируйте ее именем пакета.

`Can't localize pseudohash element`

(F) Было сказано что-то типа `local $ar->{'key'}`, где `$ar` является ссылкой на псевдохеш. Это пока не реализовано, но того же результата можно добиться, непосредственно локализовав соответствующий элемент массива: `local $ar->[$ar->[0]{'key'}]`.

`Can't locate auto/%s.al in @INC`

(F) Были вызваны функция (или метод) в пакете, допускающем автоматическую загрузку, но функции для автоматической загрузки не оказалось в наличии. Наиболее вероятная причина состоит в неправильном вводе имени функции/метода или в неудаче выполнения разбиения файла методом `AutoSplit`, например, при выполнении `make install`.

`Can't locate %s`

(F) Попытка выполнения `do` (или `require`, или `use`) для файла, который не удалось найти. Perl ищет файл по всем адресам, указанным в `@INC`, если только в имя файла не входит полный путь к нему. Возможно, следует установить переменную окружения `PERL5LIB` или `PERL5OPT`, чтобы указать, где находится дополнительная библиотека, либо сценарий должен добавить имя библиотеки к `@INC`. Либо просто неправильно введено имя файла.

`Can't locate object method "%s" via package "%s"`

(F) Вы правильно вызвали метод, и он правильно указал пакет, выступающий как класс, но в этом пакете, равно как и в его базовых классах, нет такого метода.

`Can't locate package %s for @%s::ISA`

(W syntax) Массив `@ISA` содержал имя другого пакета, по-видимому, несуществующего.

`Can't make list assignment to \%ENV on this system`

(F) Присвоение списком `%ENV` не поддерживается в некоторых системах, в частности в VMS.

`Can't modify %s in %s`

(F) Не разрешается присваивать значение указанному элементу или пытаться другим способом изменить его, например, при автоинкрементировании.

`Can't modify non-lvalue subroutine call`

(F) Подпрограммы, предназначенные для использования в качестве левого значения, должны быть соответствующим образом объявлены.

`Can't modify nonexistent substring`

(P) Внутренней программе, осуществляющей присвоение `substr`, был передан NULL.

`Can't msgrcv to read-only var`

(F) Целевой объект `msgrcv` должен быть модифицируем, чтобы он мог выступать в качестве приемного буфера.

`Can't open %s: %s`

(S inplace) Неявное открытие файла через указатель <> или явное с ключами командной строки `-n` и `-p` оказалось неудачным по указанной причине. Обычно причиной является отсутствие разрешения на чтение для файла, указанного в командной строке.

`Can't open bidirectional pipe`

(W pipe) Было сказано `open(CMD, "|cmd|")`, что не поддерживается. Можно попытаться воспользоваться любым из нескольких библиотечных модулей Perl для открытия двунаправленного канала, например `IPC::Open2`. Либо направить вывод канала в файл с помощью >, а затем читать его как другой указатель файла.

`Can't open error file %s as stderr`

(F) Специфическая для VMS ошибка. Perl осуществляет собственную переадресацию командной строки, и он не смог открыть файл, указанный после 2> или 2>> в командной строке, для записи.

`Can't open input file %s as stdin`

(F) Специфическая для VMS ошибка. Perl осуществляет собственную переадресацию командной строки, и он не смог открыть файл, указанный после <, для чтения.

`Can't open output file %s as stdout`

(F) Специфическая для VMS ошибка. Perl осуществляет собственную переадресацию командной строки, и он не смог открыть файл, указанный после > или >> в командной строке, для записи.

`Can't open output pipe (name: %s)`

(P) Специфическая для VMS ошибка. Perl осуществляет собственную переадресацию командной строки, и он не смог открыть канал, в который нужно пересылать данные, предназначенные для `STDOUT`.

`Can't open perl script "%s": %s`

(F) Заданный сценарий не мог быть открыт по указанной причине.

`Can't redefine active sort subroutine %s`

(F) Perl оптимизирует внутреннюю обработку подпрограмм сортировки и хранит указатели на них. Зафиксирована попытка переопределить одну

такую подпрограмму сортировки в то время, когда она была активной, чего делать не разрешается. Если это действительно необходимо, следует написать `sort { &func } @x`, а не `sort func @x`.

`Can't remove %s: %s, skipping file`

(S inplace) Вы потребовали редактирования по месту без создания резервной копии файла. Perl не смог удалить исходный файл, чтобы заменить его модифицированным файлом. Файл остался не модифицированным.

`Can't rename %s to %s: %s, skipping file`

(S inplace) Переименование по ключу `-i` по какой-то причине оказалось неуспешным, возможно, по причине отсутствия права записи в каталог.

`Can't reopen input pipe (name: %s) in binary mode`

(P) Ошибка, специфическая для VMS. Perl считал, что `STDIN` это канал, и попытался заново открыть его для приема двоичных данных. Увы, это не удалось.

`Can't reswap uid and euid`

(P) Вызов `setreuid` по какой-то причине оказался неуспешным в эмуляторе setuid *suidperl*.

`Can't return outside a subroutine`

(F) Команда `return` была выполнена в основном коде, т. е. не в подпрограмме, из которой можно вернуться.

`Can't return %s from lvalue subroutine`

(F) Perl обнаружил попытку возврата недопустимых значений (например, временных значений только для чтения) из подпрограммы, используемой в качестве левого значения. Это не допускается.

`Can't stat script "%s"`

(P) По каким-то причинам невозможно выполнить `fstat` для сценария, хотя он уже открыт. Странно.

`Can't swap uid and euid`

(P) Вызов `setreuid` по какой-то причине оказался неуспешным в эмуляторе setuid *suidperl*.

`Can't take log of %g`

(F) Для обычных действительных чисел нельзя логарифмировать отрицательные числа и ноль. Однако если действительно необходимо сделать это для отрицательных чисел, то с Perl поставляется стандартный пакет `Math::Complex`.

`Can't take sqrt of %g`

(F) Для обычных действительных чисел невозможна операция извлечения корня квадратного из отрицательных чисел. Однако если это действительно нужно сделать, то с Perl поставляется стандартный пакет `Math::Complex`.

`Can't undef active subroutine`

(F) Нельзя сделать неопределенной подпрограмму, выполняющуюся в данный момент. Можно, однако, переопределить ее, пока она выполняется, и даже выполнить `undef` с переопределенной подпрограммой, пока выполняется старая.

`Can't unshift`

(F) Попытка выполнить `unshift` для «не реального» массива, к которому это неприменимо, например основному стеку Perl.

`Can't upgrade that kind of scalar`

(P) Внутренняя программа `sv_upgrade` добавляет «члены» к SV, делая ее более специализированной SV. Однако несколько старших типов SV настолько специализированы, что их нельзя преобразовывать один в другой. Это сообщение указывает, что была предпринята попытка такого преобразования.

`Can't upgrade to undef`

(P) Неопределенная SV находится в самом низу схемы возможностей обновления. Обновление до `undef` указывает на ошибку в коде, вызывающем `sv_upgrade`.

`Can't use %%! because Errno.pm is not available`

(F) При первом использовании хеша `%!` Perl автоматически загружает модуль `Errno`. Модуль `Errno` должен связать хеш `%!` и обеспечить символические имена для значений ошибок `$!`.

`Can't use "my %s" in sort comparison`

(F) Глобальные переменные `$a` и `$b` зарезервированы для сравнений при сортировке. Переменные `$a` или `$b` указаны в той же строке, что и оператор `<=>` или `cmp`, и переменная была ранее объявлена как лексическая.

`Bad evalled substitution pattern`

(F) Использован ключ `/e` для вычисления замены в подстановке, но Perl обнаружил синтаксическую ошибку в коде, который нужно выполнить, скорее всего, неожиданную правую фигурную скобку `}`.

`Can't use %s for loop variable`

(F) Только простая скалярная переменная может выступать в качестве переменной цикла в `foreach`.

`Can't use %s ref as %s ref`

(F) Перепутаны типы ссылок. Необходимо разыменовать ссылку нужного типа. Для проверки типа ссылки может понадобиться использовать функцию `ref`.

`Can't use \%c to mean $%c in expression`

(W syntax) В обычном выражении обратная косая черта является унарным оператором, создающим ссылку на свой аргумент. Применение

обратного слэша для указания обратной ссылки на найденную подстроку допустимо только в составе шаблона регулярного выражения. Попытка сделать это в обычном коде Perl создает значение, которое выводится примерно как `SCALAR(0xdecaf)`. Используйте вместо этого форму `$1`.

`Can't use bareword ("%s") as %s ref while "strict refs" in use`

(F) Только жесткие ссылки допускаются при действии прагмы `strict refs`. Символические ссылки запрещены.

`Can't use string ("%s") as %s ref while "strict refs" in use`

(F) Только жесткие ссылки допускаются при действии прагмы `strict refs`. Символические ссылки запрещены.

`Can't use an undefined value as %s reference`

(F) Значение, используемое в качестве жесткой или символической ссылки, должно быть определенным. Это позволяет вскрыть некоторые коварные ошибки.

`Can't use global %s in "my"`

(F) Предпринята попытка определить «магическую» переменную в качестве лексической. Это не дозволяется, поскольку магия может привязываться только к одному адресу (а именно к глобальной переменной) и было бы крайне досадно иметь в программе переменные, выглядящие как магические, но на самом деле ими не являющиеся.

`Can't use subscript on %s`

(F) Компилятор попытался интерпретировать выражение в квадратных скобках как индекс. Но слева от скобок оказалось выражение, не напоминающее ссылку на массив или что-либо, к чему можно обратиться по индексу.

`Can't weaken a nonreference`

(F) Предпринята попытка ослабить нечто, не являющееся ссылкой. Ослаблять можно только ссылки.

`Can't x= to read-only value`

(F) Попытка повторить константу (часто неопределенное значение) в операторе присвоения, который влечет модификацию самой величины. Вероятно, следует копировать значение во временную переменную и повторить действие с ней.

`Can't find an opnumber for "%s"`

(F) Строка вида `CORE::` *word* была передана в прототип `prototype`, но встроенной функции с именем *word* нет.

`Can't resolve method `%s' overloading `%s' in package `%s'`

(F|P) Произошла ошибка во время разрешения перегрузки, заданной именем метода (в противоположность ссылке на подпрограмму): через этот пакет нельзя вызвать такой метод. Если имя метода указано как `???`, это внутренняя ошибка.

`Character class [:%s:] unknown`

(F) Класс в синтаксисе класса `[: :]` неизвестен.

`Character class syntax [%s] belongs inside character classes`

(W unsafe) Конструкции классов символов`[: :]`, `[= =]` и `[. .]` помещаются *внутри* классов символов, например: `/[012[:alpha:]345]/`. Обратите внимание, что конструкции `[= =]` и `[. .]` в данное время не реализованы; они представляют собой просто заполнители для будущих расширений.

`Character class syntax [. .] is reserved for future extensions`

(W regexp) В классах символов регулярных выражений (`[]`) синтаксис, начинающийся с `[.` и завершающийся `.]`, зарезервирован для будущих расширений. Для того чтобы представить эти последовательности символов внутри класса символов регулярного выражения, цитируйте квадратные скобки с помощью обратного слэша: `\[.` и `.\]`.

`Character class syntax [= =] is reserved for future extensions`

(W regexp) В классах символов регулярных выражений (`[]`) синтаксис, начинающийся с `[=` и завершающийся `=]`, зарезервирован для будущих расширений. Если вам нужно представить эти последовательности символов внутри класса символов регулярного выражения, цитируйте квадратные скобки с помощью обратного слэша: `\[=` и `=\]`.

`chmod() mode argument is missing initial 0`

(W chmod) Новички иногда пишут:

```
chmod 777, $filename
```

не понимая, что 777 будет интерпретироваться как десятичное число, эквивалентное восьмеричному 01411. В Perl восьмеричные константы следует предварять 0, как в C.

`Close on unopened file <%s>`

(W unopened) Попытка закрыть указатель файла, который не был открыт.

`Compilation failed in require`

(F) Perl не смог скомпилировать файл, заданный в команде `require`. Perl генерирует это общее сообщение, если обнаруженные им ошибки не были настолько тяжелыми, чтобы немедленно прекратить компиляцию.

`Complex regular subexpression recursion limit (%d) exceeded`

(W regexp) Механизм регулярных выражений использует рекурсию в сложных ситуациях, когда требуется поиск с возвращением. Глубина рекурсии ограничивается числом 32 766 или еще меньшим, если архитектура не поддерживает неограниченный рост стека. («Простые» и «средние» ситуации решаются без рекурсии, и на них ограничение не действует.) Попробуйте укоротить изучаемую строку, ввести циклы в коде Perl (например, с помощью `while`), а не в механизме регулярных выражений, или переписать регулярное выражение, чтобы оно стало проще или реже использовало возвращение.

`connect() on closed socket %s`

(W closed) Попытка соединиться с закрытым сокетом. Вы не забыли проверить значение, возвращаемое вызовом `socket`?

`Constant is not %s reference`

(F) Константа (возможно, объявленная с помощью прагмы `use constant`) разыменована, но привела к неверному типу ссылки. В сообщении указывается ожидаемый тип ссылки. Обычно это указывает на синтаксическую ошибку при разыменовании значения константы.

`Constant subroutine %s redefined`

(S|W redefine) Переопределена подпрограмма, которая ранее подлежала подстановке.

`Constant subroutine %s undefined`

(W misc) Подпрограмма, которая ранее подлежала подстановке, сделана неопределенной.

`constant(%s): %s`

(F) Анализатор обнаружил несоответствия при попытке определить перегруженную константу или при попытке найти имя, указанное в escape-последовательности `\N{...}`. Возможно, вы забыли загрузить соответствующие прагмы `overload` или `charnames`.

`Copy method did not return a reference`

(F) Метод, перегружающий `=`, является ошибочным.

`CORE:: %s is not a keyword`

(F) Пространство имен `CORE::` зарезервировано для ключевых слов Perl.

`Corrupt malloc ptr 0x%lx at 0x%lx`

(P) В пакете `malloc`, поступившем с Perl, произошла внутренняя ошибка.

`corrupted regexp pointers`

(P) Механизм регулярных выражений не смог разобраться с тем, что ему передал компилятор регулярных выражений.

`corrupted regexp program`

(P) Механизму регулярных выражений передана программа regexp без допустимого магического числа.

`Deep recursion on subroutine "%s"`

(W recursion) Эта подпрограмма вызвала себя (прямо или косвенно) на 100 раз больше, чем вернулась. Возможно, это указывает на бесконечную рекурсию, если только вы не пишете необычные программы тестирования производительности, и тогда это значит что-то другое.

`defined(@array) is deprecated`

(D deprecated) `defined` обычно бесполезно применять к массивам, потому что она проверяет неопределенность *скалярной* величины. Если требуется узнать, пуст ли массив, используйте `if (@array) { # если не пустой }`.

`defined(%hash) is deprecated`

(D deprecated) `defined` обычно бесполезно применять к хешам, потому что она проверяет неопределенность *скалярной* величины. Для того чтобы узнать, пуст ли хеш, используйте `if (%hash) { # если не пустой }`.

`Delimiter for here document is too long`

(F) В конструкции встроенного документа типа `<<FOO` метка `FOO` слишком длинна для Perl. Нужно сильно постараться, чтобы написать код, который выдает такую ошибку.

`Did not produce a valid header`

См. `Server error`.

`(Did you mean &%s instead?)`

(W) Вероятно, импортированная подпрограмма `&FOO` вызвана как `$FOO` или как-то еще.

`(Did you mean "local" instead of "our"?)`

(W misc) Помните, что `our` не локализует объявленную глобальную переменную. Она повторно объявлена в той же лексической области видимости, что излишне.

`(Did you mean $ or @ instead of %?)`

(W) Возможно, вы сказали `%hash{$key}`, имея в виду `$hash{$key}` или `@hash{@keys}`. С другой стороны, вы, может быть, просто хотели сказать `%hash` и отвлеклись.

`Died`

(F) Функции `die` передана пустая строка (эквивалентно `die ""`) или она была вызвана без аргументов, а `$@` и `$_` были пусты.

`(Do you need to predeclare %s?)`

(S) Это интеллектуальное предположение, выводимое с сообщением «*%s* found where operator expected». Часто она указывает на ссылку на подпрограмму или модуль, которые еще не объявлены. Это может быть связано с порядком кода в файле или с отсутствием команд `sub`, `package`, `require` или `use`. При ссылке на то, что еще не определено, не обязательно определять подпрограмму или пакет раньше текущего адреса. Для «предварительных» объявлений можно использовать пустые `sub foo;` или `package FOO;`.

`Document contains no data`

См. `Server error`.

`Don't know how to handle magic of type '%s'`

(P) Внутренняя обработка магических переменных проклята.

`do_study: out of memory`

(P) Это сообщение должно было быть перехвачено `safemalloc`.

`Duplicate free() ignored`

(S malloc) Внутренняя программа вызвала `free` с чем-то, что освобождено.

`elseif should be elsif`

(S) В Perl нет ключевого слова «`elseif`», потому что Ларри оно кажется уродливым. Ваш код будет интерпретирован как попытка вызвать метод `elseif` в классе, возвращенном следующим блоком. Едва ли это то, чего вы хотели.

`%s failed- -call queue aborted`

(F) Во время выполнения подпрограмм `CHECK`, `INIT` или `END` произошла не перехваченная исключительная ситуация. Обработка остатка очереди таких подпрограмм преждевременно прекращена.

`entering effective %s failed`

(F) Во время действия прагмы `use filetest` произошла неудача при переключении реальных и текущих UID или GID.

`Error converting file specification %s`

(F) Специфическая для VMS ошибка. Поскольку Perl может понадобиться работать с заданием файлов в синтаксисе VMS или Unix, он преобразует их в единую форму, когда должен действовать с ними напрямую. Либо Perl была передана недопустимая спецификация, либо это тот случай, который программы преобразования не обрабатывают. Черт побери!

`%s: Eval-group in insecure regular expression`

(F) Perl обнаружил меченые данные при компиляции регулярного выражения, содержащего утверждение нулевой ширины `(?{ ... })`, что небезопасно.

`%s: Eval-group not allowed, use re 'eval'`

(F) Регулярное выражение содержало утверждение нулевой ширины `(?{ ... })`, но эта конструкция допустима только при действии прагмы `use re 'eval'`.

`%s: Eval-group not allowed at run time`

(F) Perl пытался компилировать регулярное выражение, содержащее утверждение нулевой ширины `(?{ ... })`, на этапе исполнения, что бывает при интерполяции значений в шаблон. Это не разрешается, поскольку представляет собой угрозу защите данных. Если вы все же настаиваете, то можете сделать это, явно создав шаблон из интерполированной строки на этапе исполнения и использовав его в `eval`.

`Excessively long <> operator`

(F) Содержимое оператора `<>` не может превосходить максимальный размер идентификатора Perl. Если просто выполняется поиск по шаблону в большом списке имен файлов, можно попробовать использовать оператор `glob` или поместить имена файлов в переменную и произвести поиск в ней.

`Execution of %s aborted due to compilation errors`

(F) Последнее итоговое сообщение при неудаче компиляции Perl.

`Exiting eval via %s`

(W exiting) Имело место необычное завершение `eval`, например с помощью `goto` или команды управления циклом.

`Exiting format via %s`

(W exiting) Необычное завершение `format`, например с помощью `goto` или команды управления циклом.

`Exiting pseudoblock via %s`

(W exiting) Необычное завершение специальной блочной конструкции (такой как блок сортировки или подпрограммы), например с помощью `goto` или команды управления циклом.

`Exiting subroutine via %s`

(W exiting) Выполнение подпрограммы завершается необычным способом, например с помощью `goto` или команды управления циклом.

`Exiting substitution via %s`

(W exiting) Необычное завершение подстановки, например с помощью `return`, `goto` или команды управления циклом.

`Explicit blessing to '' (assuming package main)`

(W misc) Освящение ссылки на строку нулевой длины. В результате этого ссылка освящается в пакет `main`. Как правило, это не то, что вам требуется. Попробуйте указать целевой пакет по умолчанию, например `bless($ref, $p || 'MyPackage');`

`false [] range "%s" in regexp`

(W regexp) Диапазон в классе символов должен начинаться и оканчиваться литеральным символом, а не другим классом символов, как `\d` или `[:alpha:]`. Символ `-` в данном неверном диапазоне интерпретирован как литеральный `-`. Попробуйте цитировать `-` таким образом: `\-`.

`Fatal VMS error at %s, line %d`

(P) Специфическая для VMS ошибка. Что-то нехорошее произошло в системном сервисе или программе RTL; более подробно должно быть сказано в коде завершения Perl. Имя файла в `at %s` и номер строки в `line %d` указывают участок исходного кода Perl, при выполнении которого это произошло.

`fcntl is not implemented`

(F) На вашей машине не реализован `fcntl`. Что у вас, PDP-11?

`Filehandle %s never opened`

(W unopened) Была предпринята попытка ввода-вывода с неинициализированным указателем файла. Нужно вызвать `open` или `socket` либо конструктор из модуля `FileHandle`.

`Filehandle %s opened only for input`

(W io) Попытка записи по указателю файла, открытого только для чтения. Если необходимо, чтобы он был открыт для чтения и для записи, откройте его с +< или +>, или +>>, а не с < или ни с чем. Если вы собираетесь только писать в этот файл, используйте > или >>.

`Filehandle %s opened only for output`

(W io) Попытка чтения по указателю файла, открытого только для записи. Если необходимо, чтобы он был открыт для чтения и для записи, откройте его с +< или +>, или +>>, а не с < или ни с чем. Если вы собираетесь только писать в этот файл, используйте <.

`Final $ should be \$ or $name`

(F) Решите, для чего последним символом строки был $: это должен был быть литеральный символ доллара или имя переменной, которое почему-то отсутствует. Необходимо добавить обратную косую черту или имя.

`Final @ should be \@ or @name`

(F) Решите, для чего последним символом строки был @: это должен был быть литеральный символ «at» или имя переменной, которое почему-то отсутствует. Необходимо добавить обратную косую черту или имя.

`flock() on closed filehandle %s`

(W closed) Указатель файла, для которого выполнялась функция `flock`, оказался закрыт. Проверьте порядок выполнения команд. `flock` действует с указателями файлов. Не пытаетесь ли вы вызвать `flock` с одноименным указателем каталога?

`Format %s redefined`

(W redefine) Был переопределен формат. Чтобы подавить это предупреждение, скажите:

```
{
    no warnings;
    eval "format NAME =...";
}
```

`Format not terminated`

(F) Формат должен завершаться строкой из одной точки. Perl добрался до конца файла и не обнаружил такой строки.

`Found = in conditional, should be ==`

(W syntax) Вы сказали:

```
if ($foo = 123)
```

имея в виду:

```
if ($foo == 123)
```

(или что-то в этом роде).

gdbm store returned *%d*, errno *%d*, key "*%s*"

(S) Предупреждение из расширения GDBM_File о том, что запись закончилась неудачей.

gethostent not implemented

(F) В данной библиотеке C, очевидно, не реализован вызов gethostent, возможно, потому, что в противном случае он чувствовал бы себя морально обязанным возвратить все имена хостов в Интернете.

get*%s*name() on closed socket *%s*

(W closed) Попытка получить имя сокета для закрытого сокета. Вы не забыли проверить значение, возвращенное вызовом socket?

getpwnam returned invalid UIC %#o for user "*%s*"

(S) Предупреждение, специфическое для VMS. Вызов sys$getuai при выполнении оператора getpwnam возвратил недопустимый UIC.

getsockopt() on closed socket *%s*

(W closed) Попытка получить параметр сокета для закрытого сокета. Вы не забыли проверить значение, возвращенное вызовом socket?

glob failed (*%s*)

(W glob) Что-то произошло с внешними программами, используемыми для glob и <*.c>. Обычно это означает, что glob передан шаблон, вызвавший отказ внешней программы и завершение ее с ненулевым статусом. Если сообщение указывает, что аварийное завершение привело к выводу дампа памяти, это может также означать аварию *csh* (оболочки C). В этом случае нужно изменить все относящиеся к *csh* переменные в *config.sh*: Если установлена *tcsh*, заставьте переменные ссылаться на нее, как если бы это была *csh* (например, full_csh='/usr/bin/tcsh'); либо сделайте их все пустыми (кроме d_csh, которая должна быть 'undef'), чтобы Perl решил, что *csh* отсутствует. В любом случае после редактирования *config.sh* выполните ./*Configure -S* и перекомпилируйте Perl.

Glob not terminated

(F) Лексический анализатор нашел левую угловую скобку там, где он рассчитывал обнаружить терм, поэтому он стал искать правую угловую скобку и не нашел ее. Скорее всего, в начале строки пропущены круглые скобки, а в действительности подразумевался символ <.

Global symbol "*%s*" requires explicit package name

(F) Обнаружена директива use strict vars, указывающая, что все переменные должны иметь лексическую область видимости (с помощью my), быть объявлены заранее с помощью our или явно квалифицированы, чтобы указать пакет, в котором находится глобальная переменная (с помощью ::).

Got an error from DosAllocMem

(P) Это ошибка, специфическая для OS/2. Скорее всего, установлен Perl устаревшей версии, потому что эта ошибка не должна возникать.

`goto must have label`

(F) В отличие от `next` и `last`, не разрешается выполнять `goto` в неизвестном направлении.

`Had to create %s unexpectedly`

(S internal) Подпрограмма запросила символ из таблицы символов, который там должен был быть, но по какой-то причине его там не оказалось, и пришлось аварийным образом создать его, чтобы предотвратить дамп памяти.

`Hash %%s missing the % in argument %d of %s ()`

(D deprecated) Очень старый Perl позволял опускать `%` в именах хешей в некоторых местах. Сейчас это очень осуждается.

`Hexadecimal number > 0xffffffff non-portable`

(W portable) Заданное шестнадцатеричное число больше `2**32-1` (4 294 967 295) и потому не переносимо между системами.

`Identifier too long`

(F) Perl ограничивает идентификаторы (имена переменных, функций и т. д.) примерно 250 символами для простых имен и несколько большим числом для составных имен (типа `$A::B`). Превышены ограничения Perl. В будущих версиях Perl такие произвольные ограничения будут, вероятно, сняты.

`Ill-formed CRTL environ value "%s"`

(W internal) Специфическое для VMS предупреждение. Perl попытался прочесть внутренний массив окружения в CRTL и обнаружил элемент без знака `=`, разделяющего ключи и значения. Элемент игнорируется.

`Ill-formed message in prime_env_iter: |%s|`

(W internal) Специфическое для VMS предупреждение. Perl попытался прочесть логическое имя или определение символа CLI, готовясь к обходу `%ENV`, и не обнаружил разделителя между ключом и значением, поэтому строка проигнорирована.

`Illegal character %s (carriage return)`

(F) Perl обычно рассматривает возврат каретки в тексте программы как любые другие пробельные символы, и это значит, что Perl был скомпилирован со стандартными опциями. По каким-то причинам данная версия Perl оказалась скомпилированной без такой поддержки. Обратитесь к своему администратору Perl.

`Illegal division by zero`

(F) Попытка разделить число на 0. Либо это ошибка в логике программы, либо необходимо добавить условный оператор, чтобы уберечься от недопустимых входных данных.

`Illegal modulus zero`

(F) Попытка разделить число на 0, чтобы получить остаток. Обычно числа это не любят.

`Illegal binary digit %s`

(F) В двоичном числе есть цифра, отличная от 0 и 1.

`Illegal octal digit %s`

(F) В восьмеричном числе есть цифра 8 или 9.

`Illegal binary digit %s ignored`

(W digit) Возможно, была попытка использовать цифру, отличную от 0 и 1, в двоичном числе. Интерпретация двоичного числа прекратилась перед недопустимой цифрой.

`Illegal octal digit %s ignored`

(W digit) Возможно, была попытка использовать 8 или 9 в восьмеричном числе. Интерпретация восьмеричного числа прекратилась перед 8 или 9.

`Illegal hexadecimal digit %s ignored`

(W digit) Возможно, была попытка использовать символ, отличный от `0 ... 9`, `A ... F` или `a ... f`, в шестнадцатеричном числе. Интерпретация шестнадцатеричного числа прекращена перед недопустимым символом.

`Illegal number of bits in vec`

(F) Число бит в `vec` (третьем аргументе) должно быть степенью двух, от 1 до 32 (или 64, если это поддерживает данная платформа).

`Illegal switch in PERL5OPT: %s`

(X) Переменная окружения `PERL5OPT` может использоваться только для установки следующих ключей: `-[DIMUdmw]`.

`In string, @%s now must be written as \@%s`

(F) Раньше Perl пытался угадать, что мы собираемся интерполировать – массив или литеральное `@`. Он делал это при первом использовании строки на этапе исполнения. Теперь строки анализируются на этапе компиляции, и двусмысленные вхождения `@` должны разрешаться при помощи обратного слэша, предваряющего литерал, либо посредством объявления (или использования) массива в программе перед строкой (лексически). (Когда-нибудь он (Perl) просто будет предполагать, что `@` без обратного слэша интерполирует массив.)

`Insecure dependency in %s`

(F) Попытка сделать что-то, не понравившееся механизму меченых данных. Этот механизм включается, когда выполняется setuid или setgid, либо задан `-T` для явного его включения. Он помечает все данные, явно или косвенно полученные от пользователя, который считается недостойным доверия. Если такие данные используются в «опасной» операции, то генерируется сообщение об этой ошибке.

`Insecure directory in %s`

(F) Нельзя использовать `system`, `exec` или открытие канала в сценарии setuid или setgid, если `$ENV{PATH}` содержит каталог, в который может записывать кто угодно.

`Insecure $ENV{%s} while running %s`

(F) Нельзя применять `system`, `exec` или открытие канала в сценарии setuid или setgid, если `$ENV{PATH}`, `$ENV{IFS}`, `$ENV{CDPATH}`, `$ENV{ENV}` или `$ENV{BASH_ENV}` получены (или могут быть получены) из данных, вводимых пользователем. Сценарий должен установить для пути известное значение, используя заслуживающие доверия данные.

`Integer overflow in %s number`

(W overflow) Шестнадцатеричное, восьмеричное или двоичное число, заданное вами в виде литерала или аргумента для `hex` или `oct`, слишком большое для вашей архитектуры и поэтому превращено в число с плавающей точкой. На 32-разрядных машинах наибольшими шестнадцатеричными, восьмеричными или двоичными числами, представляемыми без переполнения, являются 0xFFFFFFFF, 037777777777 и 0b11111111111111111111111111111111 соответственно. Обратите внимание, что Perl внутренне незаметно переводит все числа в представление с плавающей точкой, что может привести к ошибкам потери точности при последующих операциях.

`Internal inconsistency in tracking vforks`

(S) Специфическое для VMS предупреждение. Perl отслеживает, сколько раз были вызваны `fork` и `exec`, чтобы определить, должен ли текущий вызов `exec` воздействовать на текущий сценарий или подпроцесс (см. «exec LIST» в *perlvms*(1)). Однако этот счетчик почему-то сбился, поэтому Perl делает предположение и воспринимает этот `exec` как запрос на завершение Perl и выполнение указанной команды.

`internal disaster in regexp`

(P) Что-то очень нехорошее произошло с анализатором регулярных выражений.

`internal urp in regexp at /%s/`

(P) Что-то пошло «наперекосяк» с анализатором регулярных выражений.

`Invalid %s attribute: %s`

(F) Указанный атрибут подпрограммы или переменной не известен Perl или пользовательскому обработчику.

`Invalid %s attributes: %s`

(F) Указанные атрибуты подпрограммы или переменной не известны Perl или пользовательскому обработчику.

`invalid [] range "%s" in regexp`

(F) В диапазоне, указанном в классе символов, минимальный символ больше максимального.

`Invalid conversion in %s: "%s"`

(W printf) Perl не понимает указанное преобразование формата.

`Invalid separator character %s in attribute list`

(F) Между элементами списка атрибутов обнаружено нечто, отличное от двоеточия и пробельного символа. Если у предыдущего атрибута был список параметров в круглых скобках, то, возможно, он закончился слишком скоро.

`Invalid type in pack: '%s'`

(F) Указанный символ не является допустимым типом упаковки.

(W pack) Указанный символ не является допустимым типом упаковки, но это молча проигнорировано.

`Invalid type in unpack: '%s'`

(F) Указанный символ не является допустимым типом распаковки.

(W unpack) Указанный символ не является допустимым типом распаковки, но это молча проигнорировано.

`ioctl is not implemented`

(F) На машине, по-видимому, не реализован `ioctl`, что довольно странно для машины, поддерживающей C.

`junk on end of regexp`

(P) Анализатор регулярных выражений в замешательстве.

`Label not found for "last %s"`

(F) Указан цикл, который нужно завершить, но в данный момент этот цикл не выполняется, даже с учетом точки возврата.

`Label not found for "next %s"`

(F) Указан цикл, который следует продолжить, но в данный момент этот цикл не выполняется, даже с учетом точки возврата.

`Label not found for "redo %s"`

(F) Указан цикл, который следует перезапустить, но в данный момент этот цикл не выполняется, даже с учетом точки возврата.

`leaving effective %s failed`

(F) Во время действия прагмы `use filetest` не удалось переключение между реальным и текущим UID или GID.

`listen() on closed socket %s`

(W closed) Ожидание соединения на закрытом сокете. Вы не забыли проверить значение, возвращенное вызовом `socket`?

`Lvalue subs returning %s not implemented yet`

(F) Из-за ограничений текущей реализации значения массив или хеш не могут возвращаться подпрограммами, используемыми в контексте левого значения.

`Malformed PERLLIB_PREFIX`

(F) Это специфическая ошибка для OS/2. `PERLLIB_PREFIX` должна иметь вид:

```
prefix1; prefix2
```

или:

```
prefix1 prefix2
```

с непустыми *prefix1* и *prefix2*. Если *prefix1* действительно является префиксом маршрута поиска библиотеки, подставляется *prefix2*. Ошибка может возникнуть, когда составляющие не найдены или слишком длинны. См. `PERLLIB_PREFIX` в файле *README.os2*, поставляемом с дистрибутивом Perl.

`Method for operation %s not found in package %s during blessing`

(F) Была сделана попытка в таблице перегрузки задать запись, которая не может быть разрешена в допустимую подпрограмму.

`Method %s not permitted`

См. `Server error`.

`Might be a runaway multi-line %s string starting on line %d`

(S) Указание на то, что предыдущая ошибка могла быть вызвана отсутствием ограничителя строки или шаблона, поскольку строка (string) фактически закончилась ранее в текущей строчке (line).

`Misplaced _ in number`

(W syntax) Подчеркивание в десятичной константе было не на границе трех цифр.

`Missing $ on loop variable`

(F) По-видимому, вы слишком много программировали для *csh*. Переменные цикла всегда указываются в Perl с `$`, в отличие от оболочек, где это может быть по-разному в разных строках.

`Missing %sbrace%s on \N{}`

(F) Неверный синтаксис литерала имени символа `\N{charname}` в контексте двойных кавычек.

`Missing comma after first argument to %s function`

(F) Хотя некоторые функции позволяют задавать указатель файла или «косвенный объект» перед списком аргументов, это не тот случай.

`Missing command in piped open`

(W pipe) Использована конструкция `open(FH, "| command")` или `open(FH, "command |")`, но команда отсутствует или пуста.

`(Missing operator before %s?)`

(S) Это интеллектуальное предположение, выводимое с сообщением «*%s* found where operator expected». Часто отсутствующим оператором является запятая.

`Missing right curly or square bracket`

(F) Лексический анализатор насчитал больше открывающих фигурных или квадратных скобок, чем закрывающих. Как правило, скобка отсутствует вблизи того места, которое редактировалось в последний раз.

`Modification of a read-only value attempted`

(F) Попытка, явная или косвенная, изменить значение константы. Конечно, вы не пытались выполнить `2 = 1`, поскольку компилятор это отлавливает. Но то же самое легко делается так:

```
sub mod { $_[0] = 1 }
mod(2);
```

Другой способ – это присвоение `substr`, вышедшей за конец строки.

`Modification of non-creatable array value attempted, subscript %d`

(F) Вы попытались косвенно создать значение массива, и индекс, вероятно, был отрицательным, даже при обратном отсчете с конца массива.

`Modification of non-creatable hash value attempted, subscript "%s"`

(P) Предпринята попытка косвенно создать значение хеша, что по какой-то причине оказалось невозможным.

`Module name must be constant`

(F) В качестве первого аргумента `use` допускается только голое имя модуля.

`msg%s not implemented`

(F) В вашей системе нет IPC-сообщений System V.

`Multidimensional syntax %s not supported`

(W syntax) Запись `$foo[1,2,3]` для многомерных массивов неправильна. Следует писать `$foo[1][2][3]`, как в C.

`Missing name in "my sub"`

(F) Зарезервированный синтаксис для подпрограмм с лексической областью видимости требует, чтобы у них было имя, по которому их можно найти.

`Name "%s:: %s" used only once: possible typo`

(W once) Ошибки ввода (опечатки) часто проявляются в виде уникальных имен переменных. Если у вас были достаточные основания для создания уникального имени, просто укажите его каким-то образом еще раз, чтобы подавить это сообщение. Для этой цели есть объявление `our`.

`Negative length`

(F) Предпринята попытка выполнить операцию `read/write/send/recv` с длиной буфера меньше 0. Это трудно себе представить.

`nested *?+ in regexp`

(F) Нельзя квантифицировать квантификатор, не привлекая скобки. Поэтому такие вещи, как `**` или `+*`, или `?*` незаконны.

Обратите внимание, что квантификаторы минимального соответствия `*?`, `+?` и `??` кажутся вложенными, но не являются таковыми.

`No #! line`

(F) Эмулятор setuid требует, чтобы в сценариях была правильно сформированная строка `#!`, даже на машинах, которые не поддерживают конструкцию `#!`.

`No %s allowed while running setuid`

(F) Некоторые операции считаются слишком ненадежными, чтобы сценарий setuid или setgid пытался их выполнить. Вообще говоря, найдется другой способ сделать то, что требуется, который если не будет безопасным, то будет, по крайней мере, достижимым.

`No -e allowed in setuid scripts`

(F) Пользователь не может задать сценарий setuid.

`No %s specified for -%c`

(F) Указанный ключ командной строки требует обязательного аргумента, но вы его не задали.

`No comma allowed after %s`

(F) В списковом операторе не должно быть запятой между переданным указателем файла или «косвенным объектом» и последующими аргументами. Иначе то, что передано, интерпретируется просто как еще один аргумент.

Это сообщение может возникать в ситуации, когда мы рассчитываем на импорт константы в наше пространство имен с помощью `use` или `import`, но такой импорт не происходит (скажем, потому, что операционная система не поддерживает эту константу). Необходим явный список импорта для констант. В этом случае данная ошибка могла быть перехвачена раньше. А может быть, просто неправильно набрано имя константы.

`No command into which to pipe on command line`

(F) Специфическая для VMS ошибка. Perl осуществляет собственную переадресацию командной строки и обнаружил `|` в конце командной строки, поэтому он не знает, куда следует направить вывод этой команды.

`No DB::DB routine defined`

(F) Выполняющийся код был скомпилирован с ключом `-d`, но по какой-то причине в файле *perl5db.pl* (или его факсимильной копии) не была определена программа, которую необходимо вызывать в начале каждой команды. Что странно, поскольку файл должен был быть затребован автоматически и вызвал бы ошибку в `require` при неудаче синтаксического анализа.

`No dbm on this machine`

(P) Это считается внутренней ошибкой; каждая машина должна сейчас предоставлять DBM, поскольку Perl поступает с SDBM.

`No DBsub routine`

(F) Выполняющийся код был скомпилирован с ключом `-d`, но по какой-то причине в файле *perl5db.pl* (или его факсимильной копии) не была определена программа `DB::sub`, которую следует вызывать в начале каждого обычного вызова подпрограммы.

`No error file after 2> or 2>> on command line`

(F) Специфическая для VMS ошибка. Perl осуществляет собственную переадресацию командной строки и обнаружил `2>` или `2>>` в командной строке, но он не может найти имя файла для записи данных, предназначенных `STDERR`.

`No input file after < on command line`

(F) Специфическая для VMS ошибка. Perl осуществляет собственную переадресацию командной строки и обнаружил `<` в командной строке, но он не может найти имя файла для чтения данных, предназначенных `STDIN`.

`No output file after > on command line`

(F) Специфическая для VMS ошибка. Perl осуществляет собственную переадресацию командной строки и обнаружил единичный `>` в конце командной строки, но не знает, куда следует переадресовать `STDOUT`.

`No output file after > or >> on command line`

(F) Специфическая для VMS ошибка. Perl осуществляет собственную переадресацию командной строки и обнаружил `>` или `>>` в командной строке, но не может найти имя файла для записи данных, предназначенных `STDOUT`.

`No package name allowed for variable %s in "our"`

(F) Полностью квалифицированные имена переменных не допускаются в объявлениях `our`, поскольку не являются осмысленными в существующей семантике. Такой синтаксис зарезервирован для будущих расширений.

`No Perl script found in input`

(F) Вызвана команда *perl -x*, но в файле не была найдена строка, начинающаяся с `#!` и содержащая слово `"perl"`.

`No setregid available`

(F) При выполнении *Configure* не нашлось чего-либо, напоминающего вызов `setregid` для данной системы.

`No setreuid available`

(F) При выполнении *Configure* не нашлось чего-либо, напоминающего вызов `setreuid` для данной системы.

`No space allowed after -%c`

(F) Аргумент для указанного ключа командной строки должен следовать сразу за ключом без пробелов между ними.

`No such pseudohash field "%s"`

(F) Предпринята попытка обратиться к массиву как к хешу, но указанное имя поля не определено. Необходимо, чтобы хеш по индексу 0 отображал все допустимые имена полей в индексы массива.

`No such pseudohash field "%s" in variable %s of type %s`

(F) Предпринята попытка обратиться к полю типизированной переменной, но в типе такого имени поля не оказалось. Поиск имен полей выполняется в хеше `%FIELDS` пакета типа во время компиляции. Хеш `%FIELDS` обычно устанавливается с помощью прагмы `fields`.

`No such pipe open`

(P) Специфическая для VMS ошибка. Внутренняя программа `my_pclose` попыталась закрыть канал, который не открывался. Раньше это должно было быть перехвачено как попытка закрыть неоткрытый канал.

`No such signal: SIG%s`

(W signal) Имя сигнала, указанное как индекс в `%SIG`, не было распознано. Введите *kill -l* в оболочке, чтобы узнать допустимые имена сигналов в своей системе.

`no UTC offset information; assuming local time is UTC`

(S) Специфическое для VMS предупреждение. Perl не смог найти смещение местного часового пояса, поэтому считает, что местное системное время эквивалентно UTC. Если это не так, определите в логическом имени `SYS$TIMEZONE_DIFFERENTIAL` число секунд, которое нужно добавить к UTC, чтобы получить местное время.

`Not a CODE reference`

(F) Perl пытался выполнить ссылку на значение, содержащее код (т. е. подпрограмму), но обнаружил, что это ссылка другого типа. Узнать, какого рода ссылка была на самом деле, можно с помощью функции `ref`.

`Not a format reference`

(F) Мы не знаем, как вам удалось создать ссылку на анонимный формат, но сообщение указывает, что вы смогли это сделать, и он не существует.

`Not a GLOB reference`

(F) Perl пытался вычислить ссылку на «typeglob» (т. е. элемент таблицы символов вида `*foo`), но обнаружил, что это ссылка другого типа. Узнать, какого рода ссылка была на самом деле, можно с помощью функции `ref`.

`Not a HASH reference`

(F) Perl пытался вычислить ссылку на хеш, но обнаружил, что это ссылка другого типа. Узнать, какого рода ссылка была на самом деле, можно с помощью функции `ref`.

`Not a perl script`

(F) Эмулятор setuid требует, чтобы в сценариях была правильно сформирована строка `#!`, даже на машинах, которые не поддерживают конструкцию `#!`. В строке должно быть слово «`perl`».

`Not a SCALAR reference`

(F) Perl пытался вычислить ссылку на скалярную величину, но обнаружил, что это ссылка другого типа. Действительный тип ссылки можно выяснить с помощью функции `ref`.

`Not a subroutine reference`

(F) Perl пытался вычислить ссылку на величину с кодом (т. е. подпрограмму), но обнаружил, что это ссылка другого типа. Действительный тип ссылки можно выяснить с помощью функции `ref`.

`Not a subroutine reference in overload table`

(F) Предпринята попытка указать запись таблицы перегрузки, в которой нет доступной подпрограммы.

`Not an ARRAY reference`

(F) Perl пытался вычислить ссылку на массив, но обнаружил, что это ссылка другого типа. Действительный тип ссылки можно выяснить с помощью функции `ref`.

`Not enough arguments for %s`

(F) Функция требует больше аргументов, чем вы задали.

`Not enough format arguments`

(W syntax) В формате указано больше полей шаблона, чем передается в следующей строке.

`Null filename used`

(F) Нельзя передавать `require` пустое имя файла, особенно потому, что на многих машинах оно означает текущий каталог!

`Null picture in formline`

(F) Первым аргументом `formline` должно быть допустимое указание шаблона формата. Аргумент оказался пустым; возможно, была передана неинициализированная величина.

`NULL OP IN RUN`

(P debugging) Некая внутренняя программа вызвала `run` с нулевым указателем кода операции.

`Null realloc`

(P) Попытка переместить NULL.

`NULL regexp argument`

(P) Внутренние программы поиска по шаблону добились большого успеха.

`NULL regexp parameter`

(P) Внутренние программы поиска по шаблону сошли с ума.

`Number too long`

(F) Perl ограничивает представление десятичных чисел в программах примерно 250 символами. Эта граница превышена. В будущих версиях Perl такие произвольные ограничения будут, вероятно, сняты, а пока воспользуйтесь научной нотацией (например, `1e6` вместо `1_000_000`).

`Octal number > 037777777777 non-portable`

(W portable) Заданное восьмеричное число больше `2**32-1` (4 294 967 295) и поэтому не переносимо между системами.

`Octal number in vector unsupported`

(F) Числа с ведущим `0` в данное время недопустимы в векторах. Интерпретация таких векторов как восьмеричных чисел, возможно, будет поддерживаться в будущих версиях.

`Odd number of elements in hash assignment`

(W misc) Задано нечетное число элементов при инициализации хеша, что странно, поскольку хеши состоят из пар ключ/значение.

`Offset outside string`

(F) Предпринята попытка выполнить операцию `read/write/send/recv` со смещением за пределами буфера. Это трудно себе представить. Единственное исключение из этого правила составляет осуществление `sysread` за пределами буфера, когда буфер расширяется и новая область заполняется нулями.

`oops: oopsAV`

(S internal) Внутреннее предупреждение, указывающее, что грамматика испорчена.

`oops: oopsHV`

(S internal) Внутреннее предупреждение, указывающее, что грамматика испорчена.

`Operation` `` `%s` `` `: no method found, %s`

(F) Попытка выполнения перегруженной операции, для которой не определен обработчик. Хотя некоторые обработчики могут автоматически генерироваться через другие обработчики, обработчика по умолчанию для любой операции нет, если только для ключа перегрузки `fallback` не задано значение «истина».

`Operator or semicolon missing before %s`

(S ambiguous) Использована переменная или вызов подпрограммы, тогда как анализатор ожидал оператор. Анализатор предположил, что в действительности пользователь хотел использовать оператор, но, скорее всего, он ошибся. Например, случайно сказанное `*foo *foo` будет интерпретировано как `*foo * 'foo'`.

`Out of memory!`

(X) Внутренняя функция Perl `malloc` возвратила 0, указывая, что оставшейся памяти (или виртуальной памяти) недостаточно для выполнения запроса. Perl не остается ничего иного, как немедленно завершить работу.

`Out of memory for yacc stack`

(F) Анализатор *yacc* хотел увеличить свой стек, чтобы продолжить анализ, но `realloc` не дала ему больше памяти, виртуальной или иной.

`Out of memory during request for %s`

(X|F) Функция `malloc` возвратила 0, указывая, что оставшейся памяти (или виртуальной памяти) недостаточно для выполнения запроса.

Запрос считался небольшим, и возможность его перехвата зависит от способа компиляции Perl. По умолчанию он не перехватывается. Однако при специальной компиляции Perl может использовать содержимое `$^M` как аварийный резерв после `die` с этим сообщением. В таком случае эта ошибка может быть *один раз* перехвачена.

`Out of memory during "large" request for %s`

(F) Внутренняя функция Perl `malloc` возвратила 0, указывая, что оставшейся памяти (или виртуальной памяти) недостаточно для выполнения запроса. Однако запрос был сочтен достаточно большим (значение по умолчанию этапа компиляции 64 Кбайт), поэтому предоставляется возможность завершения при перехвате этой ошибки.

`Out of memory during ridiculously large request`

(A) Невозможно выделить больше `2**31+` «небольшого количества» байт. Эта ошибка, скорее всего, вызвана опечаткой в программе Perl (например, `$arr[time]` вместо `$arr[$time]`).

`page overflow`

(W io) Одно обращение к `write` создало больше строк, чем умещается на странице.

`panic: ck_grep`

(P) Программа не прошла внутреннюю проверку согласованности при попытке скомпилировать `grep`.

`panic: ck_split`

(P) Программа не прошла внутреннюю проверку согласованности при попытке скомпилировать `split`.

`panic: corrupt saved stack index`

(P) У стека сохранения запрошено для восстановления больше локализованных значений, чем в нем находится.

`panic: del_backref`

(P) Программа не прошла внутреннюю проверку согласованности при попытке сбросить слабую ссылку.

`panic: die %s`

(P) Мы вытолкнули стек контекста в контекст `eval` и обнаружили, что это не контекст `eval`.

`panic: do_match`

(P) Внутренняя программа `pp_match` была вызвана с недопустимыми данными для обработки.

`panic: do_split`

(P) Что-то ужасное случилось при подготовке к `split`.

`panic: do_subst`

(P) Внутренняя программа `pp_subst` была вызвана с недопустимыми данными для обработки.

`panic: do_trans`

(P) Внутренняя программа `do_trans` была вызвана с недопустимыми данными для обработки.

`panic: frexp`

(P) Отказ библиотечной функции `frexp`, делающий невозможным выполнение `printf("%f")`.

`panic: goto`

(P) Мы вытолкнули стек контекста в контекст с заданной меткой и обнаружили, что это контекст, в котором мы не можем выполнить `goto`.

`panic: INTERPCASEMOD`

(P) Лексическому анализатору стало плохо от модификатора регистра.

`panic: INTERPCONCAT`

(P) Лексическому анализатору стало плохо при разборе строки с квадратными скобками.

`panic: kid popen errno read`

(F) Ответвленный дочерний процесс вернул непонятное сообщение с номером своей ошибки.

`panic: last`

(P) Мы вытолкнули стек контекста в контекст блока и обнаружили, что это не контекст блока.

`panic: leave_scope clearsv`

(P) Лексическая переменная с возможностью записи почему-то получила статус «только для чтения» в области видимости.

`panic: leave_scope inconsistency`

(P) Стек сохранения, по-видимому, рассинхронизировался. Во всяком случае в его вершине оказалось недопустимое `enum`.

`panic: malloc`

(P) Кто-то запросил у `malloc` отрицательное число байт.

`panic: magic_killbackrefs`

(P) Программа не прошла внутреннюю проверку согласованности при попытке сбросить все слабые ссылки на объект.

`panic: mapstart`

(P) Компилятор был сломан функцией `map`.

`panic: null array`

(P) Одной из внутренних программ массивов был передан нулевой указатель AV.

`panic: pad_alloc`

(P) Компилятор запутался, какую временную память он использовал для размещения и освобождения временных и лексических переменных.

`panic: pad_free curpad`

(P) Компилятор запутался, в какой временной памяти он осуществлял размещение и освобождение временных и лексических переменных.

`panic: pad_free po`

(P) Было внутренне обнаружено неверное смещение временной памяти.

`panic: pad_reset curpad`

(P) Компилятор запутался, в какой временной памяти он осуществлял размещение и освобождение временных и лексических переменных.

`panic: pad_sv po`

(P) Было внутренне обнаружено неверное смещение временной памяти.

`panic: pad_swipe curpad`

(P) Компилятор запутался, в какой временной памяти он осуществлял размещение и освобождение временных и лексических переменных.

`panic: pad_swipe po`

(P) Было внутренне обнаружено неверное смещение временной памяти.

`panic: pp_iter`

(P) Итератор `foreach` был вызван не в рамках контекста цикла.

`panic: realloc`

(P) Кто-то запросил отрицательное число байт у `realloc`.

`panic: restartop`

(P) Внутренняя программа запросила `goto` (или что-то аналогичное), не указав адрес.

`panic: return`

(P) Мы вытолкнули стек контекста в контекст подпрограммы или `eval` и обнаружили, что это не контекст подпрограммы или `eval`.

`panic: scan_num`

(P) Внутренняя функция Perl `scan_num` была вызвана с чем-то, не являющимся числом.

`panic: sv_insert`

(P) Программе `sv_insert` было указано удалить больше строк, чем их имелось.

`panic: top_env`

(P) Компилятор попытался выполнить `goto` или что-то столь же странное.

`panic: yylex`

(P) Лексическому анализатору стало плохо при обработке модификатора регистра.

`panic: %s`

(P) Внутренняя ошибка.

`Parentheses missing around "%s" list`

(W parenthesis) Кто-то сказал что-то вроде:

```
my $foo, $bar = @_;
when you meant:
my ($foo, $bar) = @_;
```

Запомните, что `my`, `our` и `local` связывают сильнее, чем запятая.

`Perl %3.3f required- -this is only version %s, stopped`

(F) Данному модулю требуются функции Perl более свежей версии, чем выполняемая в данный момент. А давно ли вы делали апгрейд?

`PERL_SH_DIR too long`

(F) Эта ошибка специфична для OS/2. `PERL_SH_DIR` является каталогом, содержащим оболочку *sh*. См. `PERL_SH_DIR` в файле *README.os2* дистрибутива Perl.

`Permission denied`

(F) Эмулятор setuid в *suidperl* решил, что от вас нельзя ждать ничего хорошего.

`pid %x not a child`

(W exec) Специфическое для VMS предупреждение; `waitpid` было сказано ждать процесс, не являющийся подпроцессом текущего. С точки зрения VMS все в порядке, но вам, вероятно, требовалось другое.

`POSIX getpgrp can't take an argument`

(F) В данной системе установлена `getpgrp` из POSIX, не принимающая аргументов, в отличие от версии BSD, которая принимает PID.

`Possible Y2K bug: %s`

(W y2k) Попытка выполнить конкатенацию 19 с другим числом, что может привести к проблеме 2000 года.

`Possible attempt to put comments in qw() list`

(W qw) Списки `qw` содержат элементы, разделенные пробельным символом; в литеральных строках символы комментария не игнорируются, а считаются литеральными данными. (Это могли быть и другие разделители, а не круглые скобки, приведенные здесь; часто используются также фигурные скобки.)

Вероятно, было написано нечто вроде:

```
@list = qw(
    a # комментарий
    b # еще один комментарий
);
```

хотя надо было написать так:

```
@list = qw(
    a
    b
);
```

Если комментарии действительно необходимы, создайте свой список старым способом, с кавычками и запятыми:

```
@list = (
    'a',    # комментарий
    'b',    # еще один комментарий
);
```

`Possible attempt to separate words with commas`

(W qw) Списки `qw` содержат элементы, разделенные пробельным символом, поэтому запятые не нужны для разделения элементов. (Это могли быть и другие разделители, а не круглые скобки, приведенные здесь; часто используются также фигурные скобки.)

Вероятно, было написано нечто вроде:

```
qw( a, b, c );
```

что помещает запятые в элементы списка. Запишите без запятых, если их не должно быть среди ваших данных:

```
qw( a b c );
```

`Possible memory corruption: %s overflowed 3rd argument`

(A) Вызов `ioctl` или `fcntl` вернул больше, чем просил Perl. Perl рассчитывает правильный размер буфера, но на всякий случай ставит в конце проверочный символ. Этот сторожевой байт разрушен, и Perl предполагает, что память испорчена.

`pragma "attrs" is deprecated, use "sub NAME : ATTRS" instead`

(W deprecated) Было написано что-то вроде следующего:

```
sub doit
{
    use attrs qw(locked);
}
```

Вместо этого следовало использовать новый синтаксис объявления:

```
sub doit : locked
{
    ...
```

Прагма `use attrs` является устаревшей и предоставляется только для обратной совместимости.

`Precedence problem: open %s should be open(%s)`

(S precedence) Старая неправильная конструкция:

```
open FOO || die;
```

теперь интерпретируется неверным образом как:

```
open(FOO || die);
```

из-за строгого упорядочения в грамматике Perl 5 унарных и списковых операторов. (Прежний `open` был немного тем и другим.) Следует заключить указатель файла в круглые скобки или использовать новый оператор `or` вместо `||`.

`Premature end of script headers`

См. `Server error`.

`print() on closed filehandle %s`

(W closed) Указатель файла, по которому осуществляется вывод, закрылся некоторое время тому назад. Проверьте логику программы.

`printf() on closed filehandle %s`

(W closed) Указатель файла, по которому осуществляется вывод, закрылся некоторое время тому назад. Проверьте логику программы.

`Process terminated by SIG%s`

(W) Это стандартное сообщение приложений для OS/2, в то время как приложения Unix умирают молча. Это считается особенностью переноса под OS/2. Это сообщение легко отключить с помощью соответствующих обработчиков сигналов. См. также «Process terminated by `SIGTERM/SIGINT`» в файле *README.os2* дистрибутива Perl.

`Prototype mismatch: %s vs %s`

(S unsafe) Объявляемая или определяемая подпрограмма была объявлена или определена ранее с другим прототипом функции.

`Range iterator outside integer range`

(F) Один или оба числовых аргумента оператора диапазона `..` находятся вне границ, в которых могут быть внутренне представлены целыми чис-

лами. Одна из возможностей обойти это состоит в использовании магического инкрементирования строк, с добавлением `0` перед числами.

`readline() on closed filehandle %s`

(W closed) Указатель файла, по которому производится чтение, закрылся некоторое время тому назад. Проверьте логику программы.

`realloc() of freed memory ignored`

(S malloc) Внутренняя программа вызвала `realloc` с чем-то, что уже освобождено.

`Reallocation too large: %lx`

(F) Невозможно выделить более 64 Кбайт на машине MS-DOS.

`Recompile perl with -DDEBUGGING to use -D switch`

(F debugging) Нельзя использовать опцию `-D`, если код для создания нужной выдачи не скомпилирован в Perl, что влечет некоторые накладные расходы, из-за чего это и не сделано в используемом экземпляре.

`Recursive inheritance detected in package '%s'`

(F) Свыше 100 уровней наследования. Это может указывать на непреднамеренное зацикливание в иерархии наследования.

`Recursive inheritance detected while looking for method '%s' in package '%s'`

(F) Было обнаружено свыше 100 уровней наследования при вызове метода. Это может указывать на непреднамеренное зацикливание в иерархии наследования.

`Reference found where even-sized list expected`

(W misc) Предоставлена одна ссылка, а Perl рассчитывал получить список с четным числом элементов (для присвоения хешу). Обычно это означает, что использован анонимный конструктор хеша там, где должны были быть круглые скобки. Во всяком случае для хеша требуются *пары* ключ/значение:

```
%hash = { one => 1, two => 2, };     # НЕВЕРНО
%hash = [ qw( an anon array /)];     # НЕВЕРНО
%hash = ( one => 1, two => 2, );     # правильно
%hash = qw( one 1 two 2 );           # тоже отлично
```

`Reference is already weak`

(W misc) Предпринята попытка ослабить и без того уже слабую ссылку. Такая попытка не производит никакого действия.

`Reference miscount in sv_replace()`

(W internal) Внутренняя функция `sv_replace` получила новую SV со счетчиком ссылок, отличным от 1.

`regexp *+ operand could be empty`

(F) Часть регулярного выражения с квантификатором `*` или `+` может найти пустую строку.

`regexp memory corruption`

(P) Механизм регулярных выражений не справился с тем, что ему передал компилятор регулярных выражений.

`regexp out of space`

(P) Это ошибка, которая «не может произойти», потому что `safemalloc` должна была перехватить ее раньше.

`Reversed %s= operator`

(W syntax) Оператор присвоения записан в обратном порядке. Знак `=` должен быть последним, во избежание путаницы с последующими унарными операторами.

`Runaway format`

(F) Формат содержал последовательность `~~` (повторять до пустой), но он создал 200 строк, и 200-я строка выглядела так же, как 199-я. Очевидно, не было обеспечено окончание аргументов путем применения `^` вместо `@` (для скалярных переменных) или путем сдвига или выталкивания (для массивов).

`Scalar value @%s [%s] better written as $%s [%s]`

(W syntax) Использован срез массива (заданный `@`) для выбора одного элемента массива. Как правило, лучше запросить скалярную величину (заданную `$`). Разница в том, что `$foo[&bar]` всегда ведет себя как скаляр – и в случае присваивания ей, и при вычислении ее аргумента, тогда как `@foo[&bar]` ведет себя как список в случае присваивания и предоставляет списковый контекст для индекса, что может повлечь странные последствия, когда ожидается только один индекс.

С другой стороны, если кто-то действительно надеялся работать с элементом массива как со списком, то ему следует разобраться с тем, как работают ссылки, потому что Perl не станет осуществлять для него магические преобразования между скалярами и списками.

`Scalar value @%s{%s} better written as $%s{%s}`

(W syntax) Использован срез хеша (указанный `%`) для выбора одного элемента хеша. Обычно лучше запросить скалярную величину (указанную `$`). Разница в том, что `$foo[&bar]` всегда ведет себя как скаляр, и в случае присваивания ей, и при вычислении ее аргумента, тогда как `%foo[&bar]` ведет себя как список в случае присваивания и предоставляет списковый контекст для индекса, что может повлечь странные последствия, когда ожидается только один индекс.

С другой стороны, тому, кто действительно надеялся работать с элементом хеша как со списком, следует разобраться, как работают ссылки, потому что Perl не станет осуществлять для него магические преобразования между скалярами и списками.

`Script is not setuid/setgid in suidperl`

(F) Странно, что программа *suidperl* была вызвана для сценария с не установленными битами setuid или setgid. Это имеет мало смысла.

`Search pattern not terminated`

(F) Лексический анализатор не смог обнаружить конечный разделитель конструкции // или `m{}`. Помните, что разделители в виде скобок считают уровень вложенности. Эту ошибку можно получить, опустив ведущий $ переменной `$m`.

`%sseek() on unopened file`

(W unopened) Предпринята попытка использовать функции `seek` или `sysseek` с указателем файла, который либо никогда не открывался, либо успел закрыться.

`select not implemented`

(F) На этой машине не поддерживается системный вызов `select`.

`sem%s not implemented`

(F) В системе нет семафоров IPC System V.

`semi-panic: attempt to dup freed string`

(S internal) Внутренняя программа `newSVsv` была вызвана для дублирования скаляра, который ранее помечен как свободный.

`Semicolon seems to be missing`

(W semicolon) Близлежащая синтаксическая ошибка была, возможно, вызвана отсутствием точки с запятой или другого оператора, например запятой.

`send() on closed socket %s`

(W closed) Сокет, в который направляются данные, был ранее закрыт. Проверьте логику программы.

`Sequence (? incomplete`

(F) В регулярном выражении незавершенное расширение `(?`.

`Sequence (?#... not terminated`

(F) Комментарий в регулярном выражении должен оканчиваться закрывающей круглой скобкой. Встроенные круглые скобки не допускаются.

`Sequence (?%s...) not implemented`

(F) Предлагаемое расширение регулярных выражений зарезервировало символ, но оно пока еще не написано.

`Sequence (?%s...) not recognized`

(F) Использовано расширение регулярных выражений, не имеющее смысла.

`Server error`

Это сообщение об ошибке, которое обычно отображается в окне броузера при попытке запуска программ CGI (в том числе SSI) через Web. Фактический текст ошибки может сильно отличаться от сервера к серверу. Чаще всего встречаются варианты: «`500 Server error`», «`Method` *(something)*

not permitted», «Document contains no data», «Premature end of script headers» и «Did not produce a valid header».

Это ошибка CGI, а не Perl.

Необходимо обеспечить, чтобы ваш файл был исполняемым файлом, был доступен для пользователя, от имени которого CGI выполняет сценарий (который, возможно, отличается от пользователя, под учетной записью которого вы его тестировали), не зависел от переменных окружения (например, PATH) пользователя, не выполняющего его, и не находился в месте, где сервер CGI не сможет его найти, – вот, в основном, что требуется. Дополнительные сведения посмотрите по адресам:

http://www.perl.com/CPAN/doc/FAQs/cgi/idiots-guide.html
http://www.perl.com/CPAN/doc/FAQs/cgi/perl-cgi-faq.html
ftp://rtfm.mit.edu/pub/usenet/news.answers/www/cgi-faq
http://hoohoo.ncsa.uiuc.edu/cgi/interface.html
http://www-genome.wi.mit.edu/WWW/faqs/www-security-faq.html

Следует также посмотреть Perl FAQ.

setegid() not implemented

(F) Предпринята попытка осуществить присваивание $), но операционная система не поддерживает системный вызов setegid (или его эквивалент), или, по крайней мере, *Configure* так считает.

seteuid() not implemented

(F) Попытка выполнить присваивание $>, но ваша операционная система не поддерживает системный вызов seteuid (или его эквивалент), или, по крайней мере, *Configure* так считает.

setpgrp can't take arguments

(F) В вашей системе установлен setpgrp из BSD 4.2, не принимающий аргументы, в отличие setpgid POSIX, который принимает ID процесса и ID группы процесса.

setrgid() not implemented

(F) Попытка выполнить присваивание $(, но операционная система не поддерживает системный вызов setrgid (или его эквивалент), или, по крайней мере, *Configure* так считает.

setruid() not implemented

(F) Попытка выполнить присваивание $<, но операционная система не поддерживает системный вызов setruid (или его эквивалент), или, по крайней мере, *Configure* так считает.

setsockopt() on closed socket *%s*

(W closed) Попытка установить опцию для закрытого сокета. А вы не забыли проверить значение, возвращенное вызовом socket?

`Setuid/gid script is writable by world`

(F) Эмулятор setuid не будет выполнять сценарий, правом записи в который обладает любой, потому что этот любой мог уже осуществить запись в него.

`shm%s not implemented`

(F) В вашей системе нет IPC памяти совместного доступа System V.

`shutdown() on closed socket %s`

(W closed) Попытка выполнить `shutdown` для закрытого сокета. Похоже, это излишне.

`SIG%s handler "%s" not defined`

(W signal) Обработчик сигнала, указанный в `%SIG`, на самом деле не существует. Может быть, вы поместили его не в тот пакет?

`sort is now a reserved word`

(F) Древнее сообщение об ошибке, с которым теперь почти никто не сталкивается. Но пока «sort» не стало ключевым словом, некоторые использовали его в качестве указателя файла.

`Sort subroutine didn't return a numeric value`

(F) Программа сравнения для сортировки должна возвращать число. Возможно, ошибка вызвана тем, что вы обошлись без `<=>` или `cmp` либо использовали их некорректно.

`Sort subroutine didn't return single value`

(F) Подпрограмма сравнения для сортировки не может возвращать списковое значение, в котором больше или меньше одного элемента.

`Split loop`

(P) Функция `split` вошла в бесконечный цикл. (Очевидно, `split` не должна осуществить больше итераций, чем символов во вводе, однако это и случилось.)

`Stat on unopened file <%s>`

(W unopened) Попытка использовать функцию `stat` (или эквивалентный тест файла) с указателем файла, который никогда не был открыт или с тех пор закрылся.

`Statement unlikely to be reached`

(W exec) Выполнение `exec` с командой после нее, отличной от `die`. Это почти всегда является ошибкой, поскольку `exec` возвращается только в случае неудачи. Возможно, следовало использовать `system`, которая возвращается. Чтобы подавить это предупреждение, поместите `exec` в собственный блок.

`Strange *+?{} on zero-length expression`

(W regexp) Применение квантификатора регулярного выражения в таком месте, где он бессмыслен, например, с утверждением нулевой шири-

ны. Попробуйте поместить квантификатор внутрь утверждения. Например, для поиска abc, за которым следует троекратное повторение xyz, используйте /abc(?=(?:xyz){3})/, а не /abc(?=xyz){3}/.

Stub found while resolving method `%s' overloading `%s' in package `%s'

(P) Разрешение перезагрузки через дерево @ISA может быть нарушено заглушками импортирования. Никогда не следует создавать заглушки явно, но явные обращения к can могут нарушать это.

Subroutine %s redefined

(W redefine) Переопределение подпрограммы. Для подавления этого предупреждения скажите:

```
{
    no warnings;
    eval "sub name { ... }";
}
```

Substitution loop

(P) Подстановка вошла в бесконечный цикл. (Очевидно, подстановка не должна осуществить больше итераций, чем символов во вводе, однако это и случилось.)

Substitution pattern not terminated

(F) Лексический анализатор не смог обнаружить внутренний разделитель конструкции s/// или s{}{}. Помните, что разделители в виде скобок учитывают уровень вложенности. Эту ошибку можно получить, опустив ведущий $ переменной $s.

Substitution replacement not terminated

(F) Лексический анализатор не смог обнаружить конечный разделитель конструкции s/// или s{}{}. Помните, что разделители в виде скобок учитывают уровень вложенности. Эту ошибку можно получить, опустив ведущий $ переменной $s.

substr outside of string

(W substr|F) Предпринята попытка сослаться на substr, указывавшую за пределы строки. Другими словами, абсолютное значение смещения было больше, чем длина строки. Это предупреждение является фатальным, если substr используется в контексте левого значения (как левая часть присваивания или аргумент подпрограммы, например).

suidperl is no longer needed since %s

(F) Perl был скомпилирован с -DSETUID_SCRIPTS_ARE_SECURE_NOW, но версия эмулятора setuid почему-то все равно запускалась.

switching effective %s is not implemented

(F) Во время действия прагмы use filetest нельзя переключать реальный и текущий UID или GID.

`syntax error`

(F) Это сообщение может означать синтаксическую ошибку. Частыми причинами являются:

- Ошибка в написании ключевого слова.
- Отсутствие точки с запятой.
- Отсутствие запятой.
- Пропуск открывающей или закрывающей круглой скобки.
- Пропуск открывающей или закрывающей фигурной скобки.
- Пропуск закрывающей кавычки.

Часто с синтаксической ошибкой связано другое сообщение об ошибке с дополнительной информацией. (Иногда полезно включить `-w`.) Само сообщение об ошибке часто извещает о том, в каком месте строки Perl решил сдаться. Иногда фактическая ошибка находится на несколько лексем раньше, поскольку Perl неплохо разбирает случайный ввод. Иногда номер строки может вводить в заблуждение, и в редких случаях единственный способ выяснить, что вызвало ошибку, состоит в многократном вызове *perl -c*, с отбрасыванием половины программы при каждом вызове, имеющем целью посмотреть, не исчезла ли ошибка. В общем, разновидность кибернетической версии какого-нибудь теста на сообразительность.

`syntax error at line %d: `%s' unexpected`

(A) Сценарий случайно запущен через оболочку Борна, а не Perl. Проверьте строку `#!` или сами подайте сценарий на вход Perl.

`System V %s is not implemented on this machine`

(F) Предпринята попытка что-то сделать с функцией, начинающейся с `sem`, `shm` или `msg`, но System V IPC на этой машине не реализован. (На некоторых машинах функции могут существовать, но отсутствовать в конфигурации.)

`syswrite() on closed filehandle %s`

(W closed) Указатель файла, по которому осуществляется вывод, закрылся некоторое время тому назад. Проверьте логику программы.

`Target of goto is too deeply nested`

(F) Попытка выполнения `goto` для достижения метки, глубина вложения которой слишком велика для Perl. Perl любезно отказывается.

`tell() on unopened file`

(W unopened) Попытка использования функции `tell` с указателем файла, который никогда не был открыт или с тех пор закрылся.

`Test on unopened file %s`

(W unopened) Попытка вызвать оператор тестирования файла с указателем файла, который закрыт. Проверьте логику программы.

`That use of $[ is unsupported`

(F) Присваивание `$[` сейчас строго ограничено и интерпретируется как директива компилятора. Можно только написать одно из:

```
$[ = 0;
$[ = 1;
...
local $[ = 0;
local $[ = 1;
...
```

Это сделано с целью помешать одному модулю непреднамеренно изменить основание индексов массива для другого модуля.

`The %s function is unimplemented`

Указанная функция не реализована в данной архитектуре согласно исследованиям *Configure*.

`The crypt() function is unimplemented due to excessive paranoia`

(F) *Configure* не смог найти на машине функцию `crypt`, возможно, потому, что поставщик ее не предоставил, возможно, потому, что считает, что правительство США считает ее секретной или, по крайней мере, будет продолжать делать вид, что это так.

`The stat preceding -l _ wasn't an lstat`

(F) Не имеет смысла проверять, относится ли текущий буфер stat к символической ссылке, если последняя stat, писавшая в буфер, уже прошла символическую ссылку, чтобы добраться до действительного файла. Укажите фактическое имя файла.

`This Perl can't reset CRTL environ elements (%s)`

`This Perl can't set CRTL environ elements (%s=%s)`

(W internal) Эти предупреждения специфичны для VMS. Была попытка изменить или удалить элемент внутреннего массива окружения CRTL, но экземпляр Perl компилировался с CRTL, не содержавшей внутренней функции `setenv`. Понадобится перекомпилировать Perl с CRTL, которая ее содержит, или переопределить `PERL_ENV_TABLES` (см. *perlvms*(1)), чтобы этот массив окружения не был объектом изменения `%ENV`, породившего предупреждение.

`times not implemented`

(F) В данной версии библиотеки C, очевидно, отсутствует `times`. Подозреваю, что вы работаете не в Unix.

`Too few args to syscall`

(F) Нужно передать `syscall` хотя бы один аргумент, указывающий вызываемый системный вызов.

`Too late for "-T" option`

(X) Строка `#!` (или ее местный эквивалент) в сценарии Perl содержит опцию `-T`, но Perl был вызван из командной строки без нее. Это ошибка, потому что к тому времени, когда Perl обнаруживает `-T` в сценарии, уже слишком поздно помечать что-либо из окружения. Поэтому Perl сдается.

Если сценарий Perl выполняется как команда с использованием механизма `#!` (или его местного эквивалента), эту ошибку можно обычно исправить, отредактировав строку `#!`, чтобы опция `-T` входила в первый аргумент Perl, например, изменить `perl -n -T` на `perl -T -n`.

Если сценарий Perl выполняется как *perl scriptname*, то опция `-T` должна появиться в командной строке: *perl -T scriptname*.

`Too late for "-%s" option`

(X) Строка `#!` (или ее местный эквивалент) в сценарии Perl содержит опцию `-M` или `-m`. Это ошибка, потому что опции `-M` и `-m` не предназначены для применения внутри сценариев. Используйте вместо этого объявление `use`.

`Too late to run %s block`

(W void) Блок `CHECK` или `INIT` определяется во время собственно этапа исполнения, когда возможность выполнить его уже упущена. Возможно, файл загружается с помощью `require` или `do`, а вместо этого следовало воспользоваться `use`. Либо следовало поместить `require` или `do` внутрь блока `BEGIN`.

`Too many ("s`

`Too many )"s`

(A) Сценарий случайно запущен через *csh* вместо Perl. Проверьте строку `#!` или вручную передайте свой сценарий Perl.

`Too many args to syscall`

(F) Perl поддерживает не более 14 аргументов для `syscall`.

`Too many arguments for %s`

(F) Функция требует меньше аргументов, чем указано в ее вызове.

`trailing \ in regexp`

(F) Регулярное выражение завершается обратным слэшем, не предваренным обратным слэшем. Поставьте обратный слэш.

`Transliteration pattern not terminated`

(F) Лексический анализатор не смог обнаружить внутренний разделитель конструкции `tr///` или `tr[][]`, или `y///`, или `y[][]`. Эту ошибку можно получить, опустив ведущий `$` переменных `$tr` или `$y`.

`Transliteration replacement not terminated`

(F) Лексический анализатор не смог обнаружить конечный разделитель конструкции `tr///` или `tr[][]`.

`truncate not implemented`

(F) На машине не реализован механизм усечения файлов, известный *Configure*.

`Type of arg %d to %s must be %s (not %s)`

(F) Эта функция требует, чтобы аргумент в указанной позиции имел определенный тип. Обращения к массивам должны иметь формат `@NAME` или `@{EXPR}`. А к хешам – `%NAME` или `%{EXPR}`. Недопустимо неявное разыменование – используйте формат `{EXPR}` в качестве неявного разыменования.

`umask: argument is missing initial 0`

(W umask) Маска пользователя `222` не корректна. Она должна иметь вид `0222`, поскольку восьмеричные литералы всегда начинаются в Perl с `0`, как в C.

`umask not implemented`

(F) На машине не реализована функция `umask`, а пользователь попытался ограничить права доступа для себя (`EXPR & 0700`).

`Unable to create sub named "%s"`

(F) Попытка создать подпрограмму с недопустимым именем или обратиться к такой подпрограмме.

`Unbalanced context: %d more PUSHes than POPs`

(W internal) Код завершения обнаружил внутреннюю несогласованность в числе контекстов выполнения, для которых был осуществлен вход и выход.

`Unbalanced saves: %d more saves than restores`

(W internal) Код завершения обнаружил внутреннюю несогласованность в числе значений, которые были временно локализованы.

`Unbalanced scopes: %d more ENTERs than LEAVEs`

(W internal) Код завершения обнаружил внутреннюю несогласованность в числе блоков, для которых был осуществлен вход и выход.

`Unbalanced tmps: %d more allocs than frees`

(W internal) Код завершения обнаружил внутреннюю несогласованность в числе смертных скаляров, для которых было осуществлено выделение и освобождение памяти.

`Undefined format "%s" called`

(F) Указанный формат не существует. Может быть, он в другом пакете?

`Undefined sort subroutine "%s" called`

(F) Указанная программа сравнения для сортировки не существует. Может быть, она в другом пакете?

`Undefined subroutine &%s called`

(F) Указанная подпрограмма не определена или, если она была определена, стала неопределенной.

`Undefined subroutine called`

(F) Анонимная подпрограмма, которую мы пытаемся вызвать, не определена или, если она была определена, стала неопределенной.

`Undefined subroutine in sort`

(F) Указанная программа сравнения для сортировки объявлена, но, видимо, еще не определена.

`Undefined top format "%s" called`

(F) Указанный формат, видимо, не существует. Может быть, он в действительности в другом пакете?

`Undefined value assigned to typeglob`

(W misc) Переменной typeglob присвоено неопределенное значение, например `*foo = undef`. При этом не происходит ничего. Возможно, подразумевалось `undef *foo`.

`unexec of %s into %s failed!`

(F) Вызов программы `unexec` по какой-то причине оказался неуспешным. Обратитесь к местному представителю FSF, который, возможно, и поместил ее туда.

`Unknown BYTEORDER`

(F) Нет функций для перестановки байтов для машины с таким порядком байтов.

`Unknown open() mode '%s'`

(F) Второй аргумент `open` с тремя аргументами не входит в список допустимых режимов: `<`, `>`, `>>`, `+<`, `+>`, `+>>`, `-|`, `|-`.

`Unknown process %x sent message to prime_env_iter: %s`

(P) Специфическая для VMS ошибка. Perl читал значения для `%ENV` перед итерацией по нему, и кто-то еще вставил сообщение в поток данных, ожидаемых Perl. Кто-то вконец запутался или, возможно, пытается вмешаться в то, как Perl заполняет `%ENV`, с нечестивыми целями.

`unmatched () in regexp`

(F) Круглые скобки без обратного слэша всегда должны быть парными в регулярных выражениях. Если мы работаем с *vi*, то ключ `%` может иметь ценность для нахождения парных круглых скобок.

`Unmatched right %s bracket`

(F) Лексический анализатор насчитал больше закрывающих фигурных или квадратных скобок, чем открывающих, возможно, потому, что пропущена открывающая скобка. По общему правилу, отсутствующая (если можно так сказать) скобка будет найдена вблизи того места, где в последний раз осуществлялось редактирование.

`unmatched [] in regexp`

(F) Квадратные скобки вокруг класса символов должны быть парными. Для того чтобы включить в класс символов закрывающую скобку, предпошлите ей обратную косую черту или поставьте первой.

`Unquoted string "%s" may clash with future reserved word`

(W reserved) Найдено голое слово, которое когда-нибудь может быть использовано как зарезервированное. Лучше заключить такое слово в кавычки или как-то выделить заглавными буквами, или вставить в него подчеркивание. Возможно, также, что оно объявлено как подпрограмма.

`Unrecognized character %s`

(F) У анализатора Perl нет никакого представления о том, что делать с символом, указанном в сценарии Perl (или `eval`). Возможно, вы решили запустить сжатый сценарий, двоичную программу или каталог в качестве программы Perl.

`Unrecognized escape \\%c passed through`

(W misc) Указана неизвестная Perl комбинация обратного слэша и символа.

`Unrecognized signal name "%s"`

(F) Для функции `kill` задано имя сигнала, которое не известно. Введите *kill -l* в оболочке, чтобы узнать допустимые имена сигналов в системе.

`Unrecognized switch: -%s (-h will show valid options)`

(F) Задана недопустимая опция Perl. Не делайте этого. (Если вы считаете, что не делали этого, проверьте в строке `#!`, не она ли поставляет недоброкачественный ключ от вашего имени.)

`Unsuccessful %s on filename containing newline`

(W newline) Была предпринята файловая операция по имени файла, и она оказалась неудачной либо потому, что имя файла содержало символ перевода строки, либо потому, что пользователь забыл применить к нему `chop` или `chomp`.

`Unsupported directory function "%s" called`

(F) На машине не поддерживаются `opendir` и `readdir`.

`Unsupported function fork`

(F) Данная версия исполнительного модуля не поддерживает ветвление процессов.

Обратите внимание, что в некоторых системах, например OS/2, может быть несколько видов исполняемых модулей Perl, одни из которых поддерживают `fork`, а другие – нет. Попробуйте изменить имя, под которым вы обращаетесь к Perl, на `perl_`, `perl__` и т. д.

`Unsupported function %s`

(F) По-видимому, на данной машине не реализована указанная функция. По крайней мере, так считает *Configure*.

`Unsupported socket function "%s" called`

(F) На машине не реализован механизм сокетов Berkeley, или, по крайней мере, так считает *Configure*.

`Unterminated <> operator`

(F) Анализатор обнаружил левую угловую скобку там, где рассчитывал увидеть терм, поэтому он ищет соответствующую правую угловую скобку и не находит ее. Возможно, пропущены необходимые скобки ранее в этой строке, а в действительности вы имели в виду символ `<`.

`Unterminated attribute parameter in attribute list`

(F) Лексический анализатор обнаружил символ открывающей (левой) круглой скобки при анализе списка атрибутов, но соответствующий символ закрывающей (правой) круглой скобки не был обнаружен. Возможно, потребуется добавить (удалить) обратный слэш, чтобы соблюсти баланс скобок.

`Unterminated attribute list`

(F) Лексический анализатор обнаружил нечто отличное от простого идентификатора в начале атрибута, и это не было точкой с запятой или началом блока. Возможно, слишком рано был завершен список параметров предыдущего атрибута.

`Use of $# is deprecated`

(D deprecated) Это было неблагоразумной попыткой эмулировать плохо определенную функцию *awk*. Примените вместо этого явные вызовы `printf` или `sprintf`.

`Use of $* is deprecated`

(D deprecated) Эта переменная магически активизировала поиск по шаблону в многострочном тексте – для пользователя или для несчастной подпрограммы, которую он вызвал. Сейчас следует указать модификаторы `//m` и `//s`, чтобы сделать это без опасных эффектов «действия на расстоянии» `$*`.

`Use of %s in printf format not supported`

(F) Попытка использовать формат `printf`, доступный только в C. Обычно это означает, что в Perl можно сделать лучше.

`Use of bare << to mean <<"" is deprecated`

(D deprecated) Сейчас рекомендуется явно заключенная в кавычки форма, если требуется использовать пустую строку в качестве завершения встроенного документа.

`Use of implicit split to @_ is deprecated`

(D deprecated) Вы создаете массу работы компилятору, разрушая список аргументов подпрограммы, поэтому лучше явно присвоить результаты `split` массиву (или списку).

`Use of inherited AUTOLOAD for non-method %s () is deprecated`

(D deprecated) Случайной (?) функцией являлся поиск подпрограмм `AUTOLOAD` как методов (с использованием иерархии `@ISA`), даже при вызове подлежащих автозагрузке подпрограмм как обычных функций (например, `Foo::bar()`), а не как методов (например, `Foo->bar()`или `$obj->bar()`).

Эта ошибка была исправлена в Perl 5.005, в котором применялся поиск методов только для методов `AUTOLOAD`. Однако значительная часть программ может работать в прежнем режиме. Поэтому в качестве промежуточного шага Perl 5.004 выдавал это необязательное предупреждение, когда не-методы использовали унаследованные `AUTOLOAD`.

Простое правило таково: наследование не работает при загрузке не-методов. Простое исправление для старого кода такое: в любом модуле, который зависел от наследования `AUTOLOAD` для не-методов от базового класса с именем `BaseClass`, выполняют при начальной загрузке `*AUTOLOAD = \&BaseClass::AUTOLOAD`.

В коде, в котором сказано `use AutoLoader; @ISA = qw(AutoLoader);`, следует удалить `AutoLoader` из `@ISA` и заменить `use AutoLoader;` на `use AutoLoader 'AUTOLOAD';`.

`Use of reserved word "%s" is deprecated`

(D deprecated) Указанное голое слово является зарезервированным словом. Будущие версии Perl могут распознавать его как ключевое слово, поэтому лучше явно цитировать слово способом, соответствующим контексту его употребления, или вообще указать другое имя. Это предупреждение можно подавить для имен подпрограмм, добавляя префикс `&` или используя квалификатор пакета, например `&our()`или `Foo::our()`.

`Use of %s is deprecated`

(D deprecated) Указанную конструкцию рекомендуется более не применять, в общем, потому, что существует способ сделать это лучше, и потому, что у прежнего способа были нехорошие побочные эффекты.

`Use of uninitialized value%s`

(W uninitialized) Неопределенное значение использовалось так, как если бы оно было уже определено. Оно было интерпретировано как `""` или `0`, но, возможно, это было ошибочно. Чтобы подавить это предупреждение, присвойте своим переменным определенное значение.

`Useless use of "re" pragma`

(W) Выполнение `use re` без аргументов. Это не очень полезно.

`Useless use of %s in void context`

(W void) Нечто выполнено без побочного эффекта в контексте, который ничего не делает с возвращаемым значением, например, команда, которая не возвращает значение из блока или левую часть скалярного оператора запятой. Например, мы получим эту ошибку, спутав приоритеты C с приоритетами Python и сказав:

```
$one, $two = 1, 2;
```

хотя собирались сказать:

```
($one, $two) = (1, 2);
```

Другой частой ошибкой является употребление обычных скобок для создания ссылки на список, когда необходимы были квадратные или фигурные скобки. Например, мы сказали:

```
$array = (1,2);
```

когда следовало сказать:

```
$array = [1,2];
```

Квадратные скобки явно превращают списковое значение в скалярное, а круглые – нет. Поэтому, когда список в круглых скобках вычисляется в скалярном контексте, запятая действует как оператор запятой C, отбрасывая левый аргумент, а это не то, что нам нужно.

`untie attempted while %d inner references still exist`

(W untie) Копия объекта, возвращенного `tie` (или `tied`), все еще действовала, когда была вызвана `untie`.

`Value of %s can be "0"; test with defined()`

(W misc) Использование в условном выражении `<HANDLE>`, `<*>` (glob), `each` или `readdir` в качестве булева выражения. Каждая из этих конструкций может возвратить значение `"0"`, которое сделает условное выражение ложным, что может быть не тем, что вы предполагали. При употреблении этих конструкций в условных выражениях проверяйте их значения с помощью оператора `defined`.

`Value of CLI symbol "%s" too long`

(W misc) Специфическое для VMS предупреждение. Perl попытался прочесть значение элемента `%ENV` из таблицы символов CLI и обнаружил, что результирующая строка длиннее 1024 символов. Возвращаемое значение укорочено до 1024 символов.

`Variable "%s" is not imported%s`

(F) Во время действия `use strict` мы сослались на глобальную переменную, которую, видимо, считали импортированной из другого модуля, поскольку что-то еще с тем же именем (обычно подпрограмма) экспортируется этим модулем. Обычно это значит, что мы поместили перед переменной неверный разыменовывающий символ.

`Variable "%s" may be unavailable`

(W closure) Внутренняя (вложенная) *анонимная* подпрограмма находится внутри *именованной* подпрограммы, а снаружи находится еще одна подпрограмма; при этом анонимная (самая вложенная) подпрограмма

ссылается на лексическую переменную, определенную в самой внешней подпрограмме, например:

```
sub outermost { my $a; sub middle { sub { $a } } }
```

Если к анонимной подпрограмме обращаются или ссылаются на нее (прямо или косвенно) из самой внешней подпрограммы, доступ к переменной будет совместным, как вы и рассчитывали. Но если происходит ссылка или вызов анонимной подпрограммы, когда самая внешняя подпрограмма не активна, то она будет видеть значение совместно используемой переменной таким, каким оно было до и во время *первого* вызова самой внешней подпрограммы, что, вероятно, не соответствует вашим желаниям.

В таких случаях, как правило, лучше всего сделать среднюю подпрограмму анонимной при помощи синтаксиса `sub {}`. В Perl организована особая поддержка переменных совместного доступа во вложенных анонимных подпрограммах; вставляемая именованная подпрограмма мешает выполнению этой функции.

`Variable "%s" will not stay shared`

(W closure) Внутренняя (вложенная) *именованная* подпрограмма ссылается на лексическую переменную, определенную во внешней подпрограмме.

При вызове внутренняя подпрограмма, возможно, увидит значение переменной во внешней подпрограмме таким, каким оно было до и во время *первого* вызова внешней подпрограммы; в этом случае, когда первый вызов внешней подпрограммы завершен, внутренняя и внешняя подпрограммы уже не будут использовать одно и то же значение переменной. Иными словами, переменная больше не будет общей.

Более того, если внешняя переменная анонимна и ссылается на внешнюю лексическую переменную, внутренняя и внешняя подпрограммы *никогда* не будут совместно использовать данную переменную.

Обычно эту проблему можно решить, сделав внутреннюю подпрограмму анонимной с помощью синтаксиса `sub {}`. Когда внутренние анонимные подпрограммы, которые ссылаются на переменные в других подпрограммах, вызываются или происходит ссылка на них, они автоматически заново связываются с текущими значениями таких переменных.

`Variable syntax`

(A) Сценарий случайно запущен через *csh* вместо Perl. Проверьте строку `#!` или сами вручную передайте свой сценарий Perl.

`Version number must be a constant number`

(P) Попытка оттранслировать команду *`use Module n.n LIST`* в эквивалентный блок `BEGIN` обнаружила внутреннюю несогласованность с номером версии.

`perl: warning: Setting locale failed.`

(S) (Отказ при установке национальных настроек.) В полном виде предупредительное сообщение будет выглядеть примерно так:

```
perl: warning: Setting locale failed.
    perl: warning: Please check that your locale settings:
            LC_ALL = "En_US",
            LANG = (unset)
        are supported and installed on your system.
    perl: warning: Falling back to the standard locale ("C").
```

(Могут «отказать» различные установки локализации.) Эта ошибка означает, что Perl обнаружил установленную системным администратором так называемую «переменную систему» (variable system), но Perl не смог использовать эти установки. К счастью, это не смертельно: существует «национальная настройка по умолчанию», называемая «C», которую Perl может и будет использовать, поэтому сценарий будет выполняться. Однако пока проблема не будет решена, мы будем получать это сообщение об ошибке при каждом запуске Perl. Как в действительности исправлять положение, можно прочесть в *perllocale*(1) в разделе «Locale Problems».

`Warning: something's wrong`

(W) Мы передали `warn` пустую строку (эквивалент `warn ""`) или вызвали ее без аргументов, а `$_` была пуста.

`Warning: unable to close filehandle %s properly`

(S) Неявный `close`, выполняемый `open`, получил сообщение об ошибке от `close`. Обычно это означает, что в файловой системе нет свободного места.

`Warning: Use of "%s" without parentheses is ambiguous`

(S ambiguous) Обнаружен унарный оператор, за которым следует нечто похожее на бинарный оператор, но что может также интерпретироваться как терм или унарный оператор. Например, если некто знает, что функция `rand` по умолчанию имеет аргумент `1.0` и пишет:

```
rand + 5;
```

то ему может *показаться*, что написанное равносильно

```
rand() + 5;
```

но на деле получится:

```
rand(+5);
```

Поэтому используйте скобки, чтобы высказываться яснее.

`write() on closed filehandle %s`

(W closed) Указатель файла, по которому осуществляется вывод, закрылся некоторое время тому назад. Проверьте логику программы.

`X outside of string`

(F) Шаблон `pack` указывал относительную позицию перед началом распаковываемой строки.

`x outside of string`

(F) Шаблон `pack` указывал относительную позицию после конца распаковываемой строки.

`Xsub "%s" called in sort`

(F) Применение внешней программы для сравнения при сортировке пока не поддерживается.

`Xsub called in sort`

(F) Применение внешней программы для сравнения при сортировке пока не поддерживается.

`You can't use -l on a filehandle`

(F) Указатель файла представляет открытый файл, и когда файл был открыт, он уже прошел все символические ссылки, которые мы, предположительно, ищем. Используйте имя файла.

`YOU HAVEN'T DISABLED SCRIPTS IN THE KERNEL YET!`

(F) (`ВЫ ЕЩЕ НЕ ОТКЛЮЧИЛИ СЦЕНАРИИ SET-ID В ЯДРЕ!`) И, возможно, никогда этого не сделаете, потому что у вас, возможно, нет исходного кода ядра, а поставщику его, возможно, наплевать на то, что вам нужно. Лучшим решением будет поместить C-оболочку setuid вокруг своего сценария с помощью сценария *wrapsuid* в каталоге *eg* дистрибутива Perl.

`You need to quote "%s"`

(W syntax) Голое слово присвоено в качестве имени обработчика сигнала. К несчастью, уже объявлена подпрограмма с таким именем, поэтому Perl 5 будет пытаться вызвать подпрограмму при выполнении присваивания, чего вы, вероятно, не желали. (Если это то, что вам действительно нужно, поставьте `&` спереди.)

Глоссарий

Если мы выделяем здесь курсивом слово или фразу, то это обычно означает, что можно найти их определение в этом глоссарии. Рассматривайте это как гиперссылку.

ARGV

Имя массива, содержащего *вектор аргументов* из командной строки. Когда мы применяем пустой оператор `<>`, `ARGV` служит именем *указателя файла*, используемого при проходе по аргументам, и *скаляра*, содержащего имя текущего входного файла.

ASCII

American Standard Code for Information Interchange – американский стандартный код для обмена информацией (набор 7-разрядных символов, пригодный только для ограниченного представления английского текста). Часто произвольно используется для описания младших 128 значений различных наборов символов ISO-8859-X – горстки взаимно несовместимых 8-разрядных кодов, которые лучше всего описать как полуASCII. См. также *Unicode*.

AV

Сокращение от «array value» – «значение типа массив», относящееся к одному из внутренних типов данных Perl, который содержит *массив*. Тип `AV` является подклассом *SV*.

awk

Термин, используемый при редактировании, – от «awkward» (неуклюжий). Случайно обозначает также уважаемый язык обработки текста, который послужил Perl основой некоторых его идей высокого уровня.

BLOCK

Синтаксическая конструкция, состоящая из последовательности *команд* Perl, заключенная в фигурные скобки. Команды `if` и `while`, например, определяются в терминах блоков. Иногда мы также называем «блоком» лексическую область видимости, т. е. последовательность команд, действующих как блок, например, внутри `eval` или файла, даже если команды там не заключены в фигурные скобки.

BSD

Воздействующий на психику наркотик, популярный в 80-х годах, разработанный, вероятно, в Университете Беркли или его окрестностях. Во многом аналогичен распространяемому только по рецепту лекарству «System V», но несравнимо более полезен. (Или, по крайней мере, более забавен.) Полное химическое название «Berkeley Standard Distribution».

C

Язык, который многие любят за его вывернутые наизнанку определения *типов*, непостижимые правила *приоритетов* и значительную *перегрузку* механизма вызова функций. (На самом деле, сначала на C стали переходить, обнаружив, что идентификаторы на нижнем регистре читаются легче, чем на верхнем.) Perl написан на C, поэтому не уди-

вительно, что заимствовал у него некоторые идеи.

CODE

Слово, возвращаемое функцией `ref` в случае применения ее к ссылке на подпрограмму. См. также *CV*.

CPAN

Comprehensive Perl Archive Network – архив Perl (подробности см. в предисловии и в главе 22 «CPAN»).

CV

Внутреннее определение типа «значение кода», содержащее *подпрограмму*. Тип `CV` является подклассом *SV*.

DBM

Обозначает программы «Data Base Management» – управления базами данных, эмулирующие в совокупности *ассоциативный массив* с помощью дисковых файлов. Программы используют схему динамического хеширования для нахождения любой записи с помощью двух обращений к диску. Файлы DBM позволяют программе Perl сохранять постоянный *хеш* на протяжении многих вызовов. Можно связывать переменные типа хеш с различными реализациями DBM – см. *AnyDBM_File*(3) и описание `DB_File` в главе 32 «Стандартные модули».

dweomer

Колдовство, иллюзия, фантом, фокусничество. Употребляется, когда в Perl магические результаты, которые производит *dwimmer*, делают не то, на что вы рассчитывали, а кажутся результатом таинственного колдовства и вмешательства нечистой силы. [От староанглийского]

dwimmer

DWIM служит акронимом для «Do What I Mean» – «делай, что я подразумеваю», принципа, согласно которому нечто должно выполнить то, чего вы хотите, не поднимая лишнего шума. Код, который действует таким образом. «Двимминг» может потребовать большого объема закулисной магии, которая (если она не остается надлежащим образом сокрытой) называется *dweomer*.

escape-последовательность – escape sequence

См. *метазнак*.

exec

Бросить программу текущего *процесса* и заменить ее другой, не завершая текущий процесс и не освобождая захваченные ресурсы (кроме прежнего образа в памяти).

FIFO

First In, First Out – первым пришел, первым ушел. См. также *LIFO*. Кроме того, прозвище для *именованных каналов*.

fileglob

Поиск имен файлов с помощью групповых символов («wildcard»). См. описание функции `glob`.

FMTEYEWTK

Far More Than Everything You Ever Wanted To Know – гораздо больше, чем все, что вам когда-либо требовалось узнать. Исчерпывающий трактат по какой-либо узкой теме, нечто вроде над-*FAQ*. Гораздо больше можно узнать у Тома.

GID

ID группы – в Unix числовой ID группы, который *операционная система* использует для идентификации вас и членов вашей *группы*.

glob

Строго говоря, символ `*` оболочки, который соответствует «глобу» символов, если мы пытаемся сгенерировать список имен файлов. Вольно говоря, глобов и аналогичных символов для поиска по шаблону. См. также *fileglob* и *typeglob*.

grep

Происходит от старой команды редактора Unix «Globally search for a Regular Expression and Print it» (глобальный поиск регулярного выражения и вывод его). Сейчас используется в общем смыс-

ле поиска любого вида, особенно в тексте. В Perl есть встроенная функция `grep`, которая ищет в списке элементы, удовлетворяющие любому заданному критерию, тогда как программа *grep*(1) ищет в одном или нескольких файлах строки, соответствующие *регулярному выражению*.

GV

Внутреннее определение типа «значения glob», содержащее *typeglob*. Тип `GV` является подклассом *SV*.

HV

Сокращенно от определения типа «hash value», в котором содержится внутреннее представление хеша в Perl. Тип `HV` является подклассом *SV*.

I/O

Ввод из или вывод в *файл* или *устройство*.

IO

Внутренний объект ввода/вывода. Может также обозначать *косвенный объект*.

IP

Internet Protocol, или Intellectual Property (интеллектуальная собственность).

IPC

Межпроцессное взаимодействие (Interprocess Communication).

IV

Число четыре, не путать с six, любимым редактором Тома. IV также означает внутреннее целочисленное значение типа, который может содержаться в скаляре, не путать с *NV*.

JAPH

«Just Another Perl Hacker» («еще один хакер Perl»). Умный, но таинственный фрагмент кода Perl, который в результате выполнения дает эту строку. Часто используется для иллюстрации некоторой возможности Perl, и что-то вроде непрерывного конкурса на самый непонятный Perl в конференциях Usenix.

LIFO

Last In, First Out – «последний пришел, первый ушел». См. также *FIFO*. LIFO обычно называется *стеком*.

lvaluable

Способный служить в качестве *левого значения*.

Makefile

Файл, управляющий компиляцией программы. Программам Perl обычно не нужен *Makefile*, т. к. у компилятора Perl масса самоуправления.

man

Программа Unix, которая выводит электронную документацию (страницы руководства).

monger

Сокращение от Perl Monger, поставщика Perl.

NFS

Network File System – сетевая файловая система, которая позволяет монтировать удаленную файловую систему, как если бы она была локальной.

NV

Сокращение от «Невада», никакая часть которой никогда не будет спутана с цивилизацией. NV означает также внутреннее Numeric Value (числовое значение) – внутреннее представление числа с плавающей точкой типа, который может содержаться в *скаляре*; не путать с *IV*.

pad

Сокращение от *scratchpad* – временной памяти.

PATH

Список *каталогов*, в которых система ищет программу, которую нужно *выполнить*. Этот список хранится в виде одной из *переменных окружения*, доступной в Perl как `$ENV{PATH}`.

pod

Разметка, используемая для встраивания документации в код Perl. См. главу 26.

POSIX

Спецификация переносимого интерфейса операционных систем (Portable Operating System Interface).

pp

Внутреннее сокращение для кода «push-pop», т. е. кода C, реализующего механизм стеков Perl.

pumpking

Обладатель *тыквы*.

PV

«pointer value», на внутреннем наречии Perl означает `char*`.

regex

См. *регулярное выражение*.

RFC

Request For Comment (просьба прокомментировать), что, несмотря на звучащую в названии робость, является названием ряда важных стандартизирующих документов.

root

Суперпользователь (UID == 0). Кроме того, каталог файловой системы самого верхнего уровня.

RTFM

Это произносят, когда хотят, чтобы вы прочли руководство – Read The Fine Manual.

RV

RV означает внутреннее значение ссылки (Reference Value) типа, который может содержаться в *скаляре*. Если вы еще не запутались, см. *IV* и *NV*.

script kiddie

Крэкер, который не *хакер*, но знает ровно столько, чтобы запускать сценарии. Не разбирающийся в сути (cargo-cult) программист.

sed

Уважаемый Stream Editor, у которого Perl позаимствовал некоторые свои идеи.

setgid

То же, что *setuid*, только относящееся к отдаче прав *группы*.

setuid

Обозначает программу, работающую с правами своего *владельца*, а не того, кто ее запустил (как обычно бывает). Также указывает бит в слове режима (*биты разрешений*), который управляет этой возможностью. Этот бит должен быть явно установлен владельцем, чтобы включить данную функцию, а программа должна быть аккуратно написана так, чтобы не дать больше прав, чем это необходимо.

shebang

В культуре Perl слово-гибрид, образованное из «sharp» (диез) и «bang» (восклицательный знак), обозначающее последовательность `#!`, которая сообщает системе, где найти интерпретатор.

slurp

Считать весь файл в строку за одну операцию.

STDERR

См. *стандартное устройство ошибок*.

STDIN

См. *стандартное устройство ввода*.

STDIO

См. *стандартный ввод-вывод*.

STDOUT

См. *стандартное устройство вывода*.

struct

Ключевое слово, вводящее определение или имя структуры.

SV

Сокращение для «scalar value» – скалярное значение. Но в интерпретаторе Perl любой *объект ссылки* рассматривается как член класса, производного от SV, в объектно-ориентированном порядке вещей. Каждое *значение* внутри Perl передается как указатель языка C `SV*`. *struct* SV знает свой собственный «тип объекта ссылки», а код достаточно сообразителен (надеемся), чтобы не пытаться вызывать функцию хеша с подпрограммой.

TCP

Сокращение от Transmission Control Protocol – протокол управления передачей данных. Протокол, служащий оболочкой для Internet Protocol, которая заставляет ненадежный механизм передачи пакетов выглядеть для прикладной программы как надежный *поток* байтов. (Обычно.)

TMTOWTDI

Есть более одного способа сделать это – (There's More Than One Way To Do It)[1], девиз Perl. Идея о том, что может найтись более одного допустимого способа решения рассматриваемой задачи программирования. (Это не значит, что «больше» всегда «лучше» или что все возможные пути одинаково желательны, – просто не должно быть Единственно Правильного Пути.)

troff

Почтенный наборный язык, из которого Perl взял название своей переменной `$%` и который тайно используется при наборе книг серии Camel.

typedef

Определение типа в языке C.

typeglob

Использование одного идентификатора с префиксом `*`. Например, `*name` замещает любую или все из `$name`, `@name`, `%name`, `&name` или просто `name`. Применение ее зависит от того, интерпретируется она как все или одна из них. См. раздел «Таблицы символов и указатели файлов» в главе 2.

typemap

Описание того, как типы C могут взаимно преобразовываться с типами Perl в модуле *расширения*, написанном на *XS*.

UDP

User Datagram Protocol, типичный способ отправки *датаграмм* через Интернет.

UID

ID пользователя. Часто используется в контексте владения *файлом* или *процессом*.

umask

Маска для тех *битов расширения*, которые должны быть сброшены при создании файлов и каталогов, чтобы установить политику для тех, кому мы обычно отказываем в доступе. См. функцию `umask`.

Unicode

Набор символов, включающий в основном все главные наборы символов на свете. См. *http://www.unicode.org*.

Unix

Очень большой и постоянно развивающийся язык с несколькими альтернативными и во многом несовместимыми синтаксисами, в котором каждый может определить все любым избранным им способом, и обычно так и делает. Те, кто говорит на этом языке, считают, что его легко изучить, поскольку его так легко приспособить к собственным целям, но различия в диалектах делают межплеменное общение почти невозможным, и путешественники часто ограничиваются подмножеством языка типа пиджин-инглиш. Чтобы быть понятым всюду, программист оболочки Unix должен посвятить годы освоению мастерства. Многие бегут этой кары и общаются на языке, подобном Эсперанто и называемом Perl. В древние времена Unix использовался также для обозначения некоторого кода, который пара человек из Bell Labs написали, чтобы найти применение компьютеру PDP-7, который тогда ничем особенным не занимался.

v-строка – v-string

Строка «версии» или «векторы», задаваемая как `v` с последующими десятичными целыми в точечной нотации, на-

[1] Или «Все верблюды идут в Рим». – *Примеч. ред.*

пример `v1.20.300.4000`. Каждое число преобразуется в символ с заданным порядковым значением (если чисел не менее трех, символ `v` не обязателен).

WYSIWYG

What You See Is What You Get – что видишь, то и получишь. Обычно употребляется, когда нечто, представленное на экране, соответствует тому, как оно будет выглядеть в конечном итоге, например объявления `format` в Perl. Также обозначает противоположность магическому, если все работает точно так, как оно выглядит, например, в формате `open` с тремя аргументами.

XS

Новый язык расширения для создания внешних подпрограмм (eXternal Subroutine), написанных на C или C++. Точное разъяснение см. в главе 21 «Внутри и снаружи».

XSUB

Внешняя *подпрограмма*, определенная на языке *XS*.

yacc

Yet Another Compiler Compiler – еще один компилятор компиляторов. Генератор анализаторов, без которого существование Perl было бы, вероятно, невозможно. См. файл *perly.y* в дистрибутиве исходного кода Perl.

автозагрузка – autoload

Загрузка по мере необходимости. Загружать по мере необходимости. (Также называется «отложенной» (lazy) загрузкой.) Конкретно, вызов подпрограммы `AUTOLOAD` в интересах подпрограммы, которая не определена.

автоинкрементирование – autoincrement

Автоматически добавить единицу к чему-либо, откуда оператор `++` и ведет свое название. Если вместо этого из чего-либо автоматически вычитается единица, это называется «автодекрементирование».

автоматическая генерация – autogeneration

Возможность *перегрузки операторов объектов*, при которой поведение некоторых *операторов* можно разумным образом предсказать на основе более фундаментальных операторов. При этом предполагается, что перегруженные операторы часто имеют такие же связи между собой, как обычные операторы. См. главу 13 «Перегрузка».

автоматическое расщепление – autosplit

Для автоматического расщепления строки, как это делает *ключ* (*switch*) `-a` при запуске с `-p` или `-n` для эмуляции *awk*. (См. также модуль `AutoSplit`, который никак не связан с ключом `-a`, но весьма тесно связан с автозагрузкой.)

алгоритм – algorithm

Четко определенная последовательность шагов, описанная достаточно ясно, чтобы их мог выполнить даже компьютер.

альтернативы – alternatives

Список возможных вариантов, из которых можно выбрать только один, как в вопросе: «В какую дверь вы хотите войти – A, B или C?» Альтернативы в регулярных выражениях разделяются одиночной вертикальной чертой: `|`. Альтернативы в обычных выражениях Perl разделяются двойной вертикальной чертой: `||`. Логические альтернативы в *булевых* выражениях разделяются `||` или `or`.

анонимный – anonymous

Используется для описания *объекта ссылки*, который не доступен непосредственно через именованную *переменную*. Такой объект ссылки должен быть косвенно доступен по крайней мере через одну *жесткую ссылку*. При исчезновении последней жесткой ссылки анонимный объект безжалостно уничтожается.

аргумент – argument

Некоторые данные, передаваемые *программе*, *подпрограмме*, *функции* или *ме-*

тоду и указывающие им, что они должны делать. Называются также «параметрами».

аргументы командной строки — command-line arguments

Значения, передаваемые вместе с именем программы, когда необходимо указать оболочке, что она должна выполнить команду. Эти значения передаются программе Perl через `@ARGV`.

арифметический оператор — arithmetical operator

Символ, такой как `+` или `/`, указывающий Perl на необходимость выполнения арифметических действий, которые вы должны были изучить в начальной школе.

архитектура — architecture

Тип компьютера, на котором вы работаете, где «тип» означает, что все такие компьютеры используют совместимый машинный язык. Поскольку программы Perl являются (обычно) простыми текстовыми файлами, а не исполняемыми образами, программа Perl значительно менее чувствительна к архитектуре, на которой она выполняется, чем другие языки, например C, которые компилируются в машинный код. См. также *платформа* и *операционная система*.

асинхронный — asynchronous

Касается событий или действий, относительный порядок которых во времени не детерминирован, т. к. слишком многое происходит одновременно. Поэтому асинхронное событие — это такое, о котором неизвестно, когда его следует ожидать.

ассоциативность — associativity

Определяет, какой *оператор* следует выполнить сначала — левый или правый, если есть выражение «A *оператор* B *оператор* C» и оба оператора имеют одинаковый приоритет. Операторы типа `+` являются ассоциативными слева, а операторы типа `**` — ассоциативными справа. См. в главе 3 «Унарные и бинарные операторы» список операторов с указанием их ассоциативности.

ассоциативный массив — associative array

См. *хеш*.

атом — atom

Компонент *регулярного выражения*, который может соответствовать *подстроке*, содержащей один или более символов, и рассматриваемый как неделимая синтаксическая единица любыми последующими *квантификаторами*. (В противоположность *утверждению*, которое соответствует чему-то, что имеет *нулевую ширину* и не может быть квантифицировано.)

атомарная операция — atomic operation

Когда Демокрит назвал словом «атом» неделимые частицы материи, он буквально имел в виду нечто, что нельзя разрезать: *a* (не) + *tomos* (делимое). Атомарная операция является действием, которое нельзя прервать, а не тем, которое запрещено в безъядерной зоне.

атрибут — attribute

Новая возможность, которая позволяет объявлять *переменные* и *подпрограммы* с модификаторами, например `sub foo : locked method`. Кроме того, это другое название для *переменной экземпляра объекта*.

базовый класс — base class

Общий тип «*объект*»; это означает *класс*, от которого путем *наследования* генетически производятся другие, более специфичные классы. Те, кто уважает своих предков, использует для того же название «надкласс».

байт — byte

Участок данных из восьми *битов*.

байт-код — bytecode

Гибридный язык, на котором разговаривают андроиды, когда не хотят раскрывать свою ориентацию (см. *конечник*). Получил название от аналогичных языков, на которых (по тем же причинам) общались между собой компиляторы и

интерпретаторы в конце XX века. Эти языки отличаются тем, что представляют все как последовательность байтов, не зависящую от архитектуры.

бесплатно доступное – freely available

Означает, что за это не надо платить, но копирайт для него может принадлежать кому-нибудь другому (например, Ларри).

бесплатно распространяемое – freely redistributable

Означает, что у вас не будет неприятностей с законом, если вы сделаете копию и дадите своим друзьям, а мы об этом узнаем. На самом деле мы бы хотели, чтобы вы раздали копии всем своим друзьям.

бесплатное программное обеспечение – freeware

Исторически любое программное обеспечение, которое отдают даром, особенно если при этом делают доступным и исходный код. Теперь это часто называют *программным обеспечением с открытым исходным кодом* (*open source software*). Одно время была тенденция использовать *freeware* отличительно от *open source software*, указывая только на бесплатное программное обеспечение, выпускаемое под универсальной общественной лицензией Фонда свободно распространяемого программного обеспечения (Free Software Foundation General Public License – GPL), но это трудно оправдать этимологически.

библиотека – library

Обычно представляет собой собрание процедур. В прежние времена относилась к собранию процедур в файле *.pl*. В наше время чаще относится ко всей совокупности *модулей* Perl в системе.

бинарный оператор – binary operator

Оператор, принимающий два *операнда*.

бинарный – binary

См. *двоичный*.

бит исполнения – execute bit

Специальная отметка, говорящая операционной системе, что она может выполнить эту программу. На самом деле в Unix есть три бита исполнения, а который из них используется, зависит от того, владеете ли вы файлом единолично, коллективно или вообще не владеете.

бит – bit

Целое число в диапазоне от 0 до 1 включительно. Наименьшая возможная единица хранения информации. Одна восьмая часть *байта* или доллара. (Название старого испанского доллара «piece of eight», т. е. «из восьми частей» происходит от того, что можно было разделить его на восемь частей, каждая из которых по-прежнему считалась деньгами. Поэтому для 25-центовой монеты и сегодня сохранилось название «two bits».)

битовая строка – bit string

(Битовый вектор.) Последовательность *битов*, которая в данном случае действительно рассматривается как последовательность битов.

биты разрешений – permission bits

Биты, устанавливаемые или сбрасываемые владельцем файла, чтобы разрешить или запретить доступ к нему других. Эти биты флагов являются частью режима, возвращаемого встроенной функцией `stat` при запросе относительно файла. В системах Unix можно получить дополнительные сведения на странице руководства *ls*(1).

блок, блокирование – block

То, что делает *процесс*, когда ему приходится чего-то ждать: «Мой процесс был заблокирован в ожидании диска». Не связанное с этим существительное, означающее большой фрагмент данных, имеющий размер, выбранный *операционной системой* (обычно степень двух, например 512 или 8192). Обычно обозначает данные, поступившие из дискового файла или отправляемые в него.

блок, в хешах – bucket

Область *хеш-таблицы*, состоящая (потенциально) из многих записей, ключи которых «хешируются» в одно и то же

значение в результате применения хеш-функции. (Это внутренняя политика, которая не должна вас беспокоить, если только вас не интересуют внутренности или политика.)

блочная буферизация – block buffering

Способ повышения эффективности ввода/вывода путем передачи сразу целого *блока*. По умолчанию Perl осуществляет поблочную буферизацию дисковых файлов. См. *буфер* и *буферизация команд*.

булев контекст – Boolean context

Особого рода *скалярный контекст*, используемый в условных операторах для определения истинности (или ложности) *скалярного значения*, возвращаемого выражением. Не вычисляется как строка или число. См. *контекст*.

булев – Boolean

Значение, которое является *истиной* (*true*) или *ложью* (*false*).

буфер – buffer

Область для временного хранения данных. *Блочная буферизация* означает, что данные передаются в место своего назначения, когда буфер оказывается заполненным. *Строчная буферизация* означает, что они передаются, когда получена законченная строка. *Буферизация команд* означает, что данные передаются при выполнении команды `print` (или эквивалентной ей). Если вывод не буферизован, то система обрабатывает байты по одному, не используя промежуточную область хранения. Это может оказаться весьма неэффективным.

буферизация команд – command buffering

Механизм, позволяющий накапливать вывод каждой *команды* Perl, а затем сбрасывать ее в виде одного запроса к *операционной системе*. Он включается путем установки переменной `$|` (`$AUTOFLUSH`) в истинное значение. Используется тогда, когда мы не хотим, чтобы данные задерживались и не отправлялись туда, куда положено, что может происходить, т. к. по умолчанию для *файлов* и *каналов* используется *блочная буферизация*.

буферизация строк – line buffering

Используется выходным потоком *стандартного ввода/вывода*, который очищает свой *буфер* после каждого *символа перевода строки*. Многие библиотеки стандартного ввода/вывода автоматически устанавливают буферизацию строк при выводе, направляемом на терминал.

вариадический – variadic

Относится к функции, способной принимать неопределенное число фактических аргументов.

вектор – vector

Математический жаргон для списка *скалярных значений*.

ветвление – fork

Создает дочерний *процесс*, идентичный родительскому в тот момент, когда задуман, и пока не получит собственные мысли. Поток (thread) с защищенной памятью.

взломщик – cracker

Некто, взламывающий систему безопасности в компьютерных системах. Взломщик может быть настоящим *хакером* или только *script kiddie*.

виртуальный – virtual

Обеспечивающий внешний вид чего-либо, не существующего в реальности, например, виртуальная память, не являющаяся памятью в действительности. (См. также *память*.) Противоположностью «виртуального» является «прозрачное» (transparent), что означает реальное существование чего-либо без внешнего проявления, например, прозрачная обработка Perl строк переменной длины в кодировке символов UTF-8.

висячая команда – dangling statement

Отдельная голая *команда*, без каких-либо скобок, «свисающая» с условного оператора `if` или `while`. С разрешает их использование. Perl – нет.

владелец – owner

Единственный пользователь (кроме суперпользователя), имеющий абсолют-

ный контроль над файлом. У файла может существовать *группа* пользователей, обладающих совместным владением, если это разрешено реальным владельцем. См. *биты разрешений*.

возвращаемое значение – return value

Значение, которое создает *подпрограмма* или *выражение* при вычислении. В Perl возвращаемое значение может быть *списком* или *скаляром*.

волшебство, магия – magic

Технически говоря, любая дополнительная семантика, прикрепленная к переменным, таким как `$!`, `$0`, `%ENV` или `%SIG`, либо к любой связанной переменной. Волшебные вещи происходят, когда вы модифицируете (не слишком аккуратно) эти переменные.

восьмеричный – octal

Число по основанию 8. Допустимы только цифры от 0 до 7. Восьмеричные константы начинаются в Perl с 0, например 013. См. также функцию `oct`.

временная память – scratchpad

Область, в которой конкретный вызов конкретного файла или подпрограммы хранит некоторые свои временные значения, включая переменные с лексической областью видимости.

встраивание – embedding

Когда одно содержится в другом, особенно если это необычно: «Я встроил полный интерпретатор Perl в свой редактор!»

встроенная функция – built-in

Функция, которая предопределена в языке. Даже если она скрыта в результате *переназначения*, вы всегда можете обратиться к встроенной функции, *квалифицировав* ее имя псевдопакетом `CORE::`.

встроенные документы – here documents

Названы так по аналогичной конструкции в *оболочках*, которая изображает, будто бы *строки*, следующие за *командой*, являются отдельным *файлом*, который нужно подать на вход команды, вплоть до некоторой завершающей строки. Однако в Perl это просто причудливая форма заключения в кавычки.

вызов по значению – call by value

Механизм передачи *аргументов*, при котором *формальные аргументы* ссылаются на копию *фактических аргументов*, и *подпрограмма* не может изменить фактические аргументы, изменяя формальные аргументы. См. также *вызов по ссылке*.

вызов по ссылке – call by reference

Механизм передачи *аргументов*, при котором *формальные аргументы* непосредственно ссылаются на *фактические аргументы*, и *подпрограмма* может изменять фактические аргументы, изменяя формальные аргументы. Это означает, что формальный аргумент представляет собой *псевдоним* фактического аргумента. См. также *вызов по значению*.

вызов – invocation

Действие, направленное на вызов божества, демона, программы, метода, подпрограммы или функции, имеющее целью заставить их делать то, что, по вашему мнению, они должны делать. Обычно «call» относится к подпрограммам, а «invoke» к методам, поскольку это звучит более модно.

вызывающий - invocant

Агент, от имени которого вызывается *метод*. Для метода *класса* вызывающим (инвокантом) является имя пакета. Для метода *экземпляра* вызывающей является ссылка на объект.

выполнять – execute

Запустить текущую *программу* или *подпрограмму*. (Не имеет отношения к встроенной функции `kill`, если только вы не пытаетесь запустить *обработчик сигнала*.)

выражение – expression

Все, что допускается сказать там, где требуется *значение*. Обычно составляется из *литералов*, *переменных*, *операторов*, *функций* и вызовов *подпрограмм*, не обязательно в указанном порядке.

высокомерие – hubris

Чрезмерная гордость, из разряда вещей, за которые Зевс мечет в нас свои молнии. Также качество, заставляющее нас писать (и сопровождать) такие программы, о которых другие не смогут сказать ничего худого. Посему является третьей великой добродетелью программиста. См. также *лень* и *нетерпение*.

генератор кода – code generator

Система, которая пишет для вас код на языке низкого уровня, например, код, реализующий сервер компилятора. См. *генератор программ*.

генератор программ – program generator

Система, алгоритмически создающая для пользователя код на языке высокого уровня. См. также *генератор кода*.

глобальное разрушение – global destruction

Уборка мусора для глобальных элементов (и выполнение всех связанных с этим деструкторов объектов), которая происходит по завершении работы *интерпретатора* Perl. Глобальное разрушение не следует путать с Апокалипсисом, исключая тот случай, когда они совпадают.

глобальный – global

Нечто, что видно отовсюду; обычно используется в отношении *переменных* и *подпрограмм*, которые видны в любом месте программы. В Perl подлинно глобальными являются лишь несколько специальных переменных – большинство переменных (и все подпрограммы) существуют только в текущем *пакете*. Глобальные переменные могут объявляться с помощью `our`. См. раздел «Глобальные объявления» главы 4 «Команды и объявления».

голое (простое) слово – bareword

Слово достаточно неоднозначное, для того чтобы рассматриваться как незаконное, если действует директива `use strict 'subs'`. В отсутствие такого ограничения голое слово рассматривается, как если бы оно было заключено в кавычки.

группа – group

Множество пользователей, которому вы принадлежите. В некоторых операционных системах (например, Unix) вы можете давать определенные права доступа к файлам остальным членам своей группы.

дамп памяти – core dump

Тело *процесса* в виде файла, сохраненного в *рабочем каталоге* процесса, обычно как результат некоторых видов фатальных ошибок.

датаграмма – datagram

Пакет данных, например сообщение *UDP*, которое (с точки зрения участвующих программ) может независимо пересылаться по сети. (На самом деле все пакеты посылаются независимо на уровне *IP*, но *поточные* протоколы, например *TCP*, скрывают это от вашей программы.)

двоичный – binary

Относящийся к числам по основанию 2. Это означает, что, по существу, есть два числа, 0 и 1. Употребляется также для описания «не текстового» файла, возможно, потому, что такой файл использует все двоичные разряды своих байтов. С приходом *Unicode* такое различие, и без того подозрительное, еще более утрачивает свое значение.

декрементирование – decrement

Вычитание значения из переменной, например, «декрементировать `$x`» (что означает вычитание 1 из значения) или «декрементировать `$x` на 3».

дерево синтаксического анализа – parse tree

См. *синтаксическое дерево*.

дескриптор файла – file descriptor

Маленькое число, используемое *операционной системой*, чтобы следить, о каком открытом *файле* идет речь. Perl скрывает дескриптор файла внутри потока *стандартного ввода-вывода*, а за-

тем прикрепляет поток к *указателю файла*.

дескриптор – descriptor

См. *дескриптор файла*.

деструктор – destructor

Особый *метод*, вызываемый тогда, когда *объект* собирается *разрушить* себя. Метод `DESTROY` ничего фактически не разрушает; Perl просто запускает этот метод, когда *класс* нуждается в какой-нибудь чистке, связанной с разрушением.

динамическая область видимости – dynamic scoping

Динамическая область видимости делает переменные видимыми во всей оставшейся части *блока*, в котором они впервые использованы, и в любых *подпрограммах*, вызываемых оставшейся частью блока. Значения переменных с динамической областью видимости могут временно изменяться (и неявно восстанавливаться позже) с помощью оператора `local`. (Сравните с *лексической областью видимости*.) Более вольно употребляется, обозначая, что подпрограмма в середине вызова другой подпрограммы «содержит» эту последнюю на *этапе исполнения*.

директива – directive

Директива *pod*. См. главу 26 «Документация в формате pod».

дистрибутив – distribution

Стандартный, увязанный выпуск системы программного обеспечения. Обычное использование предполагает наличие исходного текста. Если это не так, дистрибутив называется двоичным (binary-only).

доступа методы – accessor methods

Метод предназначен для косвенного просмотра или обновления состояния *объекта* (его *переменных экземпляра*).

единица компиляции – compilation unit

Файл (или *строка* в случае `eval`), компилируемый в данный момент.

жадный – greedy

Подшаблон, *квантификатор* которого хочет найти как можно больше.

жесткая ссылка – hard reference

Скалярная величина, содержащая фактический адрес *объекта ссылки*, которая учитывается счетчиком *ссылок* объекта ссылки. (Некоторые жесткие ссылки хранятся внутренне, например, неявная ссылка в одной из позиций переменной *typeglob* на соответствующий объект ссылки.) Жесткая ссылка отличается от *символической ссылки*.

заглядывание вперед – lookahead

Утверждение, которое заглядывает в строку правее текущего найденного соответствия.

заглядывание назад – lookbehind

Утверждение, которое заглядывает в строку левее текущего найденного соответствия.

заголовочный файл – header file

Файл, содержащий некие определения, которые необходимо включить «во главе» своей программы, чтобы произвести некоторые скрытые операции. Заголовочный файл C имеет расширение *.h*. Perl в действительности не имеет заголовочных файлов, хотя исторически он иногда использовал оттранслированные файлы *.h* с расширением *.ph*. См. `require` в главе 29 «Функции». (Заголовочные файлы заменены механизмом *модулей*.)

замещение – overriding

Сокрытие или отключение некоторого другого определения с тем же именем. (Не путать с *перегрузкой*, которая добавляет определения, двусмысленность которых должна быть разрешена каким-то другим способом.) Чтобы еще сильнее запутать вопрос, мы используем слово с двумя перегруженными определениями: для описания того, как можно определить собственную *подпрограмму* и скрыть встроенную *функцию* с таким же именем (см. раздел «Замещение встроенных функций» главы 11 «Модули»),

и для описания того, как можно определить в *производном классе* замещающий *метод*, чтобы скрыть метод *базового класса* с тем же именем (см. главу 12).

замыкание – cloister

Кластер, используемый для ограничения области действия *модификатора регулярного выражения*.

замыкание – closure

Анонимная подпрограмма, которая, когда ссылка на нее создается на этапе исполнения, продолжает отслеживать идентичность видимых снаружи *лексических переменных* даже после того, как эти лексические переменные должны были выйти из *области видимости*. Название «замыкание» дано в связи с тем, что подобное их поведение дает математикам ощущение замыкания.

запись – record

Набор взаимосвязанных значений данных в *файле* или *потоке*, часто связываемый с уникальным полем *ключа*. В Unix часто соответствует строке или группе строк, оканчивающейся пустой строкой («параграфу»). Каждая строка файла */etc/passwd* представляет собой запись, содержащую сведения о пользователе, ключом которой служит имя регистрации.

зарезервированные слова – reserved words

Слова с особым встроенным значением для *компилятора*, например `if` или `delete`. Во многих языках (но не в Perl) не допускается употребление зарезервированных слов для чего-либо другого. (В конце концов, потому они и зарезервированы.) В Perl нельзя использовать их только в *метках* и *указателях файлов*. Также называются «ключевыми словами».

захват – capturing

Применение круглых скобок для выделения *подшаблонов* в *регулярных выражениях* с целью сохранить найденную *подстроку* как *ссылку на найденный текст* (*backreference*). (Захваченные строки возвращаются в виде списка в *списковом контексте*.)

зернистость – granularity

Говоря абстрактно, размер фрагментов, с которыми вы работаете.

значение по умолчанию – default

Значение, которое используется, если не задано никакое иное.

значение – value

Фактические данные, в отличие от переменных, ссылок, ключей, индексов, операторов и всего прочего, что нужно для доступа к значению.

зомби – zombie

Процесс, который умер (завершился), но родители которого не получили надлежащего извещения о его кончине посредством вызова `wait` или `waitpid`. При выполнении ветвления `fork` следует осуществлять уборку за своими дочерними процессами после их завершения, иначе таблица процессов переполнится и ваш системный администратор будет вами недоволен.

идентификатор – identifier

Сформированное в соответствии с правилами имя, применяемое почти для всего, что может интересовать компьютерную программу. Многие языки (в том числе Perl) допускают идентификаторы, которые начинаются с буквы и содержат буквы и цифры. Perl считает также буквой символ подчеркивания. (В Perl есть также более сложные имена, например *квалифицированные* имена.)

именованный канал – named pipe

Канал с именем, встроенным в *файловую систему*, к которому поэтому могут иметь доступ два несвязанных *процесса*.

импорт – import

Получение доступа к символам, экспортируемым другим модулем. См. `use` в главе 29.

имя команды – command name

Имя программы, выполняемой в данный момент, как оно было введено в ко-

мандной строке. В C имя *команды* передается как первый аргумент командной строки. В Perl оно содержится в переменной `$0`.

имя файла – filename

Это имя перечислено в *каталоге* и может использоваться функцией `open`, когда мы сообщаем *операционной системе* о том, какой именно файл мы собираемся открыть, и связываем файл с *указателем файла*, с которым и будет ассоциироваться этот файл в нашей программе, пока мы не закроем его.

индекс – subscript

Значение, указывающее на позицию конкретного *элемента* в *массиве*.

индексация – indexing

В прежние времена состояла в поиске *ключа* в фактическом справочнике (например, телефонной книге), но сейчас это просто применение любого ключа или позиции для отыскания соответствующего *значения*, даже если никакого индекса (указателя) не существует. Дело дошло до того, что функция `index` в Perl просто находит позицию (индекс) одной строки в другой.

инкапсуляция – encapsulation

Завеса абстракции, отделяющая интерфейс от реализации (вынужденная или нет), требующая, чтобы любой доступ к состоянию *объекта* осуществлялся только через *методы*.

инкрементирование – increment

Увеличение значения чего-либо на 1 (или другое число, если оно указано).

интерполяция переменных – variable interpolation

Интерполяция переменной скаляра или массива в строку.

интерполяция – interpolation

Вставка скалярного или спискового значения в середину другого значения, так что кажется, будто оно было там всегда. В Perl интерполяция переменных осуществляется в двойных кавычках и шаблонах, а интерполяция списков происходит при создании списка значений для передачи списковому оператору или другой конструкции, принимающей список.

интерпретатор – interpreter

Строго говоря, это программа, которая читает вторую программу и делает то, что эта вторая программа говорит, непосредственно, не преобразуя сначала эту программу в другую форму, чем занимаются *компиляторы*. Perl не является интерпретатором согласно этому определению, поскольку он содержит некий компилятор, который берет программу и преобразует ее в более выполняемую форму (*синтаксические деревья*) внутри самого процесса *perl*, которую затем интерпретирует система *этапа исполнения* Perl.

интерфейс – interface

Сервисы, которые некий код обещает предоставлять всегда, в отличие от *реализации*, которую он может изменить, когда того пожелает.

инфиксный оператор – infix

Оператор, который располагается между своими *операндами*, например умножение: `24 * 7`.

исключительная ситуация – exception

Причудливый термин, обозначающий ошибку. См. *фатальная ошибка*.

исполняемый файл – executable file

Файл, помеченный особым образом, говорящим операционной системе о том, что его можно выполнять как программу.

истина – true

Любое скалярное значение, которое отлично от 0 или `""`.

исходный каталог – home directory

Каталог, в который система помещает пользователя после регистрации. В системах Unix его имя часто помещается в `$ENV{HOME}` или `$ENV{LOGDIR}` при регистрации, но можно найти его и посредством `(getpwuid($<))[7]`. (На некоторых платформах нет понятия исходного каталога.)

итератор – iterator

Специальная программная штуковина, которая следит за тем, в каком месте того, что вы пытаетесь обойти, вы находитесь. Цикл `foreach` в Perl содержит итератор, как и хеш, что позволяет применить к нему `each`.

итерация – iteration

Повторяющееся выполнение чего-либо.

канал – pipe

Прямое *соединение*, которое передает выход одного *процесса* на вход другого без посредничества временного файла. Когда канал установлен, соответствующие два процесса могут осуществлять чтение и запись, как в обычный файл, с некоторыми предостережениями.

канонический – canonical

Приведенный к стандартной форме для облегчения сравнения.

каталог – directory

Особый файл, который содержит другие файлы. В некоторых *операционных системах* такие файлы могут называться «папками», «ящиками» и т. д.

квалифицированный – qualified

С указанием полного имени. Символ `$Ent::moot` квалифицирован; `$moot` не квалифицирован. Полностью квалифицированное имя задается от каталога верхнего уровня.

квантификатор – quantifier

Компонент *регулярного выражения*, указывающий число повторений предшествующего *атома*.

класс символов – character class

Список символов, заключенный в квадратные скобки и указывающий в *регулярных выражениях* на то, что любой символ из набора может находиться в данном месте. Более широко – любой предопределенный набор символов, используемый таким образом.

класс – class

Определенный пользователем *тип*, реализованный в Perl через *пакет*, который предоставляет (непосредственно или в результате наследования) *методы* (т. е. *подпрограммы*) для работы с *экземплярами* класса (его *объектами*). См. также *наследование*.

кластер ключей – switch cluster

Объединение нескольких ключей командной строки (например, `-a -b -c`) в один ключ (т. е. `-abc`). Любой ключ с дополнительным *аргументом* должен быть последним в кластере.

кластер – cluster

Заключенный в круглые скобки *подшаблон*, используемый для объединения частей *регулярного выражения* в единый *атом*.

клиент – client

В сетевом взаимодействии – *процесс*, который инициирует контакт с процессом *сервера*, чтобы обменяться данными и, возможно, получить обслуживание.

ключ – key

Строковый индекс *хеша*, используемый для поиска *значения*, связанного с этим ключом.

ключ – switch

Опция, передаваемая в командной строке с целью повлиять на ход выполнения программы, обычно предваряемая знаком «минус». Слово может также использоваться как кличка для *переключателя*.

ключевое слово – keyword

См. *зарезервированные слова*.

команда управления циклом – loop control statement

Любая команда в теле цикла, которая может досрочно завершить цикл или пропустить *итерацию*. Как правило, лучше не пробовать выполнить это на американских горах.

команда – command

В программировании *оболочки* означает синтаксическое сочетание имени программы с ее аргументами. В более свободном применении – любые данные,

введенные в оболочку (командный процессор), заставившие ее что-то сделать. Еще более вольно – это *команда* Perl, которая может начинаться с *метки* и обычно заканчивается точкой с запятой.

команда – statement

Команда компьютеру относительно того, что делать дальше, как действие в рецепте: «Добавить мармелад в тесто и перемешивать, пока не перемешается». Команда отличается от *объявления*, которое указывает компьютеру не сделать что-либо, а узнать что-либо.

комментарий – comment

Ремарка, которая не оказывает влияния на смысл программы. В Perl комментарии начинаются с символа # и продолжаются до конца строки.

компилятор – compiler

Строго говоря, это программа, которая «пережевывает» другую прогамму и «выплевывает» файл, содержащий программу в «более выполняемом» виде, обычно в виде родных машинных инструкций. Программа *perl*, согласно этому определению, не является компилятором, но в ней содержится некий компилятор, который принимает программу и преобразует ее в более выполняемую форму (*синтаксические деревья*) внутри самого процесса *perl*, которую затем интерпретирует *интерпретатор*. Однако имеются *модули* расширения, которые заставляют Perl действовать более похожим на «настоящий» компилятор образом. См. главу 18 «Компиляция».

конвейер – pipeline

Ряд *процессов*, соединенных *каналами*, когда каждый процесс передает свой выходной поток следующему.

конец файла – EOF

Конец файла. Иногда метафорически используется как строка-терминатор *встроенного документа*.

конечник – endian

См. *остроконечник* и *тупоконечник*.

конкатенация – concatenation

Процедура, в ходе которой нос одного кота приклеивается к хвосту другого, а также аналогичная операция для двух *строк*.

конструирование – construct

Создание *объекта* с помощью *конструктора*.

конструктор – constructor

Любой *метод класса*, *метод экземпляра* или *подпрограмма*, которые создают, инициализируют, освящают и возвращают *объект*. Иногда мы вольно используем это термин для обозначения *формирователя*.

конструкция – construct

Синтаксический элемент, составленный из более мелких деталей.

контекст массива – array context

Архаическое выражение, обозначающее то, что правильнее называть *списковым контекстом*.

контекст – context

Окружение или среда. Контекст, заданный окружающим кодом; определяет, какого типа данные конкретное *выражение*, как предполагается, должно возвратить. Три основные контекста следующие: *списковый контекст*, *скалярный контекст* и *пустой контекст*. Скалярный контекст иногда подразделяется на *булев контекст*, *числовой контекст*, *строковый контекст* и *пустой контекст*. Есть и «безразличный» контекст (о котором, если вам интересно, говорится в главе 2 «Всякая всячина»).

контрольная точка – breakpoint

Место программы, в котором отладчику приказано остановить *выполнение*, чтобы можно было осмотреться и выяснить, не произошло ли чего-нибудь неправильного.

контрольное выражение – watch expression

Выражение, при изменении значения которого происходит остановка в точке прерывания в отладчике Perl.

косвенность – indirection

Если нечто в программе представляет собой не значение, которое нам нужно, а указывает на местонахождение этого значения, то это косвенность. Она может осуществляться с помощью *жестких ссылок* или *символических ссылок*.

косвенный объект – indirect object

В английской грамматике так называют короткий оборот с существительным между глаголом и его прямым дополнением, указывающий на того, к кому относится действие (косвенное дополнение). В Perl `print STDOUT "$foo\n";` можно понять как «глагол косвенное-дополнение дополнение», где `STDOUT` является получателем действия `print`, а `"$foo"` – выводимым объектом. Аналогично при вызове *метода* можно поместить вызывающего между методом и его аргументами:

```
$gollum = new Pathetic::Creature "Smeagol";
give $gollum "Fisssssh!";
give $gollum "Precious!";
```

косвенный указатель файла – indirect filehandle

Выражение, значение которого можно использовать в качестве *указателя файла*: *строка* (имя указателя файла), *typeglob*, ссылка на typeglob или *объект ввода-вывода* низкого уровня.

левое значение – lvalue

Термин, используемый законниками языка для адреса памяти, которому можно присваивать новые *значения*, например, *переменная* или *массив*. Буква «l» означает «left», как в левой части присваивания, типичного места для левых значений. Функция или выражение типа *lvaluable* означает те из них, которым могут присваиваться значения, как, например, в `pos($x) = 10`.

лексема – lexeme

Иное название для *маркера* (*token*).

лексема – token

Морфема в языке программирования, наименьшая единица текста с семантической значимостью.

лексическая область видимости – lexical scoping

Рассматривание *Oxford English Dictionary* в микроскоп. (Известна также как *статическая область видимости*, поскольку словари меняются медленно.) Аналогично рассматривание переменных, хранимых в личном словаре (пространстве имен), для каждой области видимости, которые видны только от точки объявления до конца лексической области видимости, в которой они определены. Синоним – *статическая область видимости*. Антоним – *динамическая область видимости*.

лексическая переменная – lexical variable

Переменная, на которую действует *лексическая область видимости*, объявленная с помощью `my`. Часто называется просто «lexical». (Ключевое слово `our` объявляет переменную с лексической областью видимости для глобальной переменной, которая сама не является лексической переменной.)

лексический анализ – lexical analysis

Иное название для *разбиения на лексемы* (*tokenizing*).

лексический анализатор – lexer

Иное название для *tokener*.

лексический анализатор – tokener

Модуль, который разбивает текст программы в последовательность *лексем* для последующего анализа синтаксическим анализатором.

лень – laziness

Качество, заставляющее совершать большие усилия, чтобы сократить общий расход энергии. Оно заставляет писать экономящие труд программы, которые окажутся полезными другим людям, и документировать то, что вы написали, чтобы не пришлось отвечать на слишком много вопросов о них. Посему первая великая добродетель программиста. Посему и эта книга. См. также *нетерпение* и *высокомерие*.

литерал – literal

Лексема в языке программирования, например, число или *строка*, которые дают фактическое *значение*, а не просто представляют возможные значения, как это делает *переменная*.

логический оператор – logical operator

Символы, представляющие понятия «и», «или», «исключающее или» и «не».

ложь – false

В Perl это любое значение, которое выглядит как `""` или `"0"`, если вычислено в строковом контексте. Поскольку неопределенные значения вычисляются как `""`, все неопределенные значения ложны, но не все ложные значения являются неопределенными.

локальный – local

Не всегда означает одно и то же. Глобальная переменная Perl может быть локализована внутри *динамической области видимости* посредством оператора `local`.

магические переменные – magical variables

Специальные переменные с побочными эффектами, проявляющимися при обращении к ним или присваивании. Например, в Perl изменение элементов массива `%ENV` изменяет соответствующие переменные окружения, которые будут использоваться подпроцессами. Чтение переменной `$!` дает номер текущей системной ошибки или сообщение для нее.

магическое инкрементирование – magical increment

Оператор инкрементирования, который умеет наращивать буквы так же, как и цифры.

массив – array

Упорядоченная последовательность *значений*, хранимых таким образом, что к любому из них можно легко обратиться по *целочисленному индексу*, задающему *смещение* значения в последовательности.

метазнак – metasymbol

Нечто, что мы назвали бы *метасимволом*, за исключением того, что это последовательность из более чем одного символа. Обычно первым символом этой последовательности является настоящий метасимвол, который заставляет остальные символы метазнака тоже вести себя особым образом.

метасимвол – metacharacter

Символ, обращение с которым *не* обычно. Какие именно символы должны восприниматься особым образом как метасимволы, – в значительной мере зависит от контекста. В каждой конкретной *оболочке* имеются некоторые свои метасимволы, заключенные в двойные кавычки, *строки* Perl имеют другие метасимволы, а шаблоны *регулярных выражений* имеют все метасимволы двойных кавычек плюс некоторые свои собственные.

метка цикла – loop label

Некий ключ или имя, закрепленные за циклом, чтобы команды управления циклом могли указывать, которым циклом они хотят управлять.

метка – label

Имя, которое мы даем команде, чтобы можно было говорить о ней в любом месте программы.

метод класса – class method

Метод, для которого вызов производится от имени *пакета*, а не ссылки на *объект*. Метод, ассоциируемый с классом в целом.

метод – method

Определенного рода действие, которое *объект* может выполнить, если сказать ему об этом. См. главу 12 «Объекты».

меченый – tainted

Относится к данным, полученным из грязных рук пользователя, на которые поэтому не может полагаться надежная программа. Perl осуществляет проверку меченых данных, если выполняется

программа *setuid* (или *setgid*) либо если указан ключ -T.

минимализм – minimalism

Вера в то, что «малое прекрасно». Парадоксально, но если сказать что-то в маленьком языке, оно становится большим, а если сказать что-то в большом языке, оно становится маленьким. Решайте сами.

многомерный массив – multidimensional array

Массив с несколькими индексами для нахождения одного элемента. Perl реализует их с помощью ссылок – см. главу 9 «Структуры данных».

множественное наследование – multiple inheritance

Черты, приобретенные от матери и отца, смешанные непредсказуемым образом. (См. *наследование* и *одиночное наследование*.) В языках программирования (включая Perl) представление о том, что у данного класса может быть несколько прямых предков, или *базовых классов*.

модификатор команды – statement modifier

Условие или *цикл*, помещаемые после *команды*, а не перед ней, если вы понимаете, о чем мы говорим.

модификатор левого значения – lvalue modifier

Адъективированная (играющая роль прилагательного) псевдофункция, которая искажает смысл некоторого *левого значения* некоторым декларативным образом. В настоящее время есть три модификатора левых значений: `my`, `our` и `local`.

модификатор регулярного выражения – regular expression modifier

Опция для шаблона или подстановки, например `/i`, для придания шаблону нечувствительности к регистру. См. также *замыкание – cloister*.

модификатор – modifier

См. *модификатор команды*, *модификатор регулярного выражения* и *модификатор левого значения*, не обязательно в указанном порядке.

модуль – module

Файл, который определяет *пакет* с (почти) тем же именем, который может либо *экспортировать* символы, либо функционировать как класс *объекта*. (Главный файл модуля *.pm* может также загружать другие файлы для поддержки модуля.) См. встроенную функцию `use`.

модуль – modulus

Целочисленный делитель, если нас интересует остаток, а не частное.

мягкая ссылка – soft reference

См. *символическая ссылка*.

на проходе – en passant

Когда значение изменяется во время копирования. [от французского «на проходе», как в необычном маневре взятия пешки в шахматах]

надкласс – superclass

См. *базовый класс*.

наследование – inheritance

То, что мы получаем от своих предков – генетически или иным способом. Если нечто представляет собой *класс*, то предки этого нечто называются *базовыми классами*, а потомки – *производными классами*. См. *одиночное наследование* и *множественное наследование*.

нетерпение – impatience

Гнев, который вы испытываете, когда компьютер ведет себя лениво. Это заставляет писать программы, которые не просто реагируют на наши потребности, но предвосхищают их. Или по крайней мере делают вид. Посему является второй великой добродетелью программиста. См. также *лень* и *высокомерие*.

номер ошибки – errno

Номер ошибки, возвращаемый при неудаче *системного вызова*. Perl ссылается на ошибку через `$!` (или `$OS_ERROR`, если вы используете модуль English).

номер строки – line number

Число строк, прочитанных перед данной, плюс 1. Perl ведет отдельную нумерацию строк для каждого открытого им исходного или входного файла. Текущий номер строки файла исходного текста представлен в `__LINE__`. Номер текущей строки ввода (для файла, который последним читался через `<FH>`) представляется переменной `$.` (`$INPUT_LINE_NUMBER`). Многие сообщения об ошибках содержат оба значения, если они доступны.

носильщик – porter

Некто, «переносящий» программное обеспечение с одной *платформы* на другую. Перенос программ, написанных на платформозависимых языках, таких как C, может быть трудным делом, но перенос таких программ, как Perl, стоит перенесенных мучений.

нулевая строка – null string

Строка, не содержащая символов; не путать со строкой, содержащей *нулевой символ*, которая имеет положительную длину и является истинной.

нулевая ширина – zero width

Утверждение в подшаблоне, соответствующее *нулевой строке* между *символами*.

нулевой символ – null character

Символ со значением ASCII «ноль». Используется в C для завершения строк, но в Perl разрешается, чтобы строка содержала символы null.

нулевой список – null list

Списковое значение, в котором ноль элементов, представляемое в Perl посредством `()`.

область видимости – scope

Насколько издалека видна переменная. В Perl есть два механизма видимости. *Динамическая область видимости переменных* `local` означает, что оставшаяся часть *блока* и любые *подпрограммы*, вызываемые в оставшейся части блока, могут видеть переменные, локальные для блока. *Лексическая область видимости* переменных `my` означает, что оставшаяся часть блока может видеть переменную, но другие подпрограммы, вызываемые блоком, *не* могут видеть переменную.

оболочка – shell

Интерпретатор командной строки. Программа, которая выдает интерактивное приглашение, принимает одну или несколько *строчек* ввода и выполняет указанные программы, передавая каждой из них надлежащие *аргументы* и входные данные. Оболочки могут также выполнять сценарии, состоящие из таких команд. В Unix в число часто используемых оболочек входят оболочка Борна (*/bin/sh*), оболочка C (*/bin/csh*) и оболочка Корна (*/bin/ksh*). В строгом смысле Perl не является оболочкой, поскольку не интерактивен (хотя программы Perl могут быть интерактивны).

оболочка – wrapper

Программа или подпрограмма, которая запускает вместо нас некоторую другую программу или подпрограмму, частично модифицируя ее входные и выходные данные для лучшего соответствия нашим задачам.

обработка исключительных ситуаций – exception handling

Реакция программы на возникновение ошибки. Механизм обработки исключительных ситуаций в Perl реализуется оператором `eval`.

обработчик сигнала – signal handler

Подпрограмма, которая вместо того, чтобы удовлетвориться тем, что ее вызовут обычным образом, сидит и ждет грома среди ясного неба, чтобы соизволить *выполниться*. В Perl гром среди ясного неба носит название сигнала, а посылаются они с помощью встроенной функции `kill`. См. хеш `%SIG` в главе 28 в раз-

деле «Специальные имена» и раздел «Сигналы» главы 16 «Межпроцессное взаимодействие».

обработчик – handler

Подпрограмма или *метод*, которые вызываются Perl, когда программа должна ответить на какое-то внутреннее событие, например, *сигнал* или обнаружение оператора, подлежащего *перегрузке*. См. также *обратный вызов*.

обратная совместимость – backward compatibility

Означает, что вы по-прежнему можете выполнять старую программу, поскольку мы не устранили функции или ошибки, на которых она основывалась.

обратный вызов – callback

Обработчик, который вы регистрируете в какой-либо части программы в надежде, что другая часть программы запустит его при возникновении некоторого события, представляющего интерес.

общественное владение – public domain

Нечто, владельцем чего не является никто. Perl защищен авторским правом и не находится в общественном владении – он просто *бесплатно доступен* и *бесплатно распространяется*.

объект ссылки – referent

То, на что указывает ссылка, обладающее именем (или не обладающее). Обычными объектами ссылок являются скаляры, массивы, хеши и подпрограммы.

объект – object

Экземпляр класса. Нечто, «знающее», что собой представляет определенный пользователем тип (класс) и что оно может делать согласно этому классу. Программа может попросить объект что-то сделать, а объект будет решать, хочет он это делать или нет. Одни объекты более любезны, чем другие.

объявление – declaration

Утверждение о том, что нечто существует; возможно, описывающее это нечто, но не содержащее никаких указаний на время, место и способ применения этого нечто. Объявление – это как часть рецепта, где сказано: «две чашки муки, одно большое яйцо, четыре-пять головастиков...». Противоположность *объявлению – команда*. Обратите внимание, что некоторые объявления функционируют одновременно как команды. Объявление подпрограммы действует также как определение, если прилагается ее тело.

ограничитель – delimiter

Символ или *строка*, устанавливающие границы текстового объекта произвольного размера, которые не надо путать с *разделителем* (*separator*) или *терминатором* (*terminator*). «Ограничивать» в действительности значит «окружать» или «заключать» (подобно данным круглым скобкам).

одиночное наследование – single inheritance

Черты, которые некто приобрел от матери, если она сообщила ему, что у него нет отца. (См. также *наследование* и *множественное наследование*.) В языках программирования представление о том, что *классы* размножаются неполовым путем, поэтому у данного класса может быть только один прямой предок, или *базовый класс*. Perl не накладывает такого ограничения, хотя, конечно, программировать на Perl можно и в таком стиле, если есть желание.

однострочник – one-liner

Целая компьютерная программа, втиснутая в одну строку текста.

окружение – environment

Совокупность *переменных окружения*, унаследованных *процессом* от родителя. Доступ осуществляется через `%ENV`.

операнд – operand

Выражение, дающее *значение*, к которому применяется *оператор*. См. также *приоритет*.

оператор отношения – relational operator

Оператор, который сообщает, является ли *истинным* для пары *операндов* неко-

торое конкретное отношение порядка. В Perl есть операторы отношения для чисел и строк. См. *сортирующая последовательность*.

оператор прямого доступа к памяти – address operator

Некоторые языки работают непосредственно с адресами величин в памяти, но это игра с огнем. Perl предоставляет комплект асбестовых перчаток для работы с памятью. В Perl ближе всего к оператору адреса стоит оператор обратной косой черты, но он дает *жесткую ссылку*, что значительно безопаснее, чем адрес памяти.

оператор тестирования файла – file test operator

Встроенный унарный оператор, используемый для проверки истинности какого-либо утверждения о файле; например `-o $filename` проверяет, являетесь ли вы владельцем файла.

оператор – operator

Штука, которая преобразует некоторое число входных значений в некоторое число выходных значений, часто встраиваемая в язык с помощью специального синтаксиса или символа. Этот самый оператор может иметь определенные ожидания относительно того, какие *типы* данных мы передаем ему в качестве аргументов (*операндов*) и какого типа данные собираемся получить от него обратно.

операционная система – operating system

Особая программа, которая выполняется на голой машине и скрывает детали управления *процессами* и *устройствами*. Обычно употребляется в более вольном смысле, обозначая определенную культуру программирования. Вольный смысл может интерпретироваться на разных уровнях конкретности. Одной крайностью будет сказать, что все версии Unix- и Unix-подобных систем суть одна и та же операционная система (очень многих этим расстроив, особенно законоведов и адвокатов). Другая крайность – сказать, что данная конкретная версия операционной системы данного конкретного поставщика отличается от всех других версий операционных систем любых поставщиков. Perl значительно лучше переносится между операционными системами, чем многие другие языки. См. также *архитектура* и *платформа*.

определенный – defined

Имеющий значение. Perl полагает, что некоторые вещи из тех, которые люди пытаются делать, лишены смысла, в частности, использование переменных, которым не присвоено значение, и осуществление некоторых действий над данными, которых нет. Например, при попытке чтения за концом файла Perl возвращает неопределенное значение. См. также *false* и оператор `defined` в главе 29 «Функции».

опции – options

См. *ключи* или *модификаторы регулярных выражений*.

освящение – bless

В жизни корпораций означает предоставление официального одобрения чего-либо, например: «Вице-президент по инжинирингу благословил наш проект WebCruncher». Аналогично, в Perl означает предоставление официального одобрения на то, чтобы *объект ссылки* мог функционировать как *объект*, например объект WebCruncher. См. описание функции `bless` в главе 29 «Функции».

остроконечник – little-endian

Из Свифта: тот, кто разбивает яйцо с острого конца. Употребляется также в отношении компьютеров, которые хранят менее значащий *байт* слова по более младшему адресу байта, чем более значащий байт. Часто считается, что такие компьютеры превосходят «тупоконечные» машины. См. также *тупоконечник*.

отправлять – dispatch

Посылать что-либо в правильное место. Часто употребляется метафорически,

означая передачу программного управления в точку, выбранную алгоритмически, часто путем поиска в таблице *ссылок* на функции или, в случае методов *объекта*, при обходе дерева наследования в поисках наиболее узкого определения метода.

охватывающий оператор – circumfix operator

Оператор, который охватывает свой *операнд*, подобно оператору угловых скобок или круглым скобкам, или объятиям.

очистка буфера – flush

Действие, состоящее в опустошении *буфера*, часто при его переполнении.

ошибка – error

См. *исключительная ситуация* или *фатальная ошибка*.

пакет – package

Пространство имен для глобальных *переменных*, *подпрограмм* и тому подобного, предназначенное для того, чтобы хранить их отдельно от одноименных *символов* в других пространствах имен. В некотором смысле только пакет является глобальным, т. к. символы в таблице символов пакета доступны из кода, скомпилированного вне пакета, только через указание имени пакета. Но в другом смысле все символы пакета являются также глобальными – это просто хорошо организованные глобальные символы.

память совместного доступа – shared memory

Участок *памяти*, к которому имеют доступ два разных *процесса*, которые в ином случае не имели бы возможности видеть память друг друга.

память – memory

Это всегда означает оперативную память, а не диск. Затемняет проблему то, что на машине может быть реализована *виртуальная* память, т. е. машина будет делать вид, что у нее больше памяти, чем есть на самом деле, используя дисковое пространство для хранения неактивных битов. При этом может показаться, что памяти несколько больше, чем есть на самом деле, но она не является заменой для настоящей памяти. Самое хорошее, что можно сказать о виртуальной памяти, это то, что она позволяет производительности падать постепенно, а не внезапно, когда кончается память настоящая. Но программа может умереть, если у вас закончится и виртуальная память тоже, если раньше вы не загоните свой диск до смерти.

параметр – parameter

См. *аргумент*.

патч – patch

«Заплатка» – исправление путем наложения. В хакерстве это список различий между двумя версиями программы, который можно применить с помощью программы *patch*(1) при необходимости исправить ошибку или обновить старую версию.

перевод строки – newline

Одиночный символ, представляющий конец строки, со значением ASCII «восьмеричное 012» в Unix (но 015 в Mac) и представляемый метазнаком `\n` в строках Perl. Для машин Windows, выводящих тестовые файлы, и некоторых физических устройств типа терминалов отдельный символ перевода строки автоматически транслируется библиотекой C в перевод строки и возврат каретки, но обычно никакой трансляции не производится.

перегрузка операторов – operator overloading

Вид *перегрузки*, который можно производить для встроенных *операторов*, чтобы заставить их работать с *объектами* так, как если бы они были обычными скалярными значениями, но с фактической семантикой, предоставляемой классом объекта. Перегрузка устанавливается *прагмой* `overload` – см. главу 13.

перегрузка – overloading

Сообщение дополнительного смысла символу или конструкции. Фактически

любые языки в той или иной мере производят перегрузку, поскольку человек умеет определять смысл из *контекста*.

переключатель – switch statement

Прием программирования, который позволяет вычислить *выражение*, а затем, основываясь на его значении, осуществить переход к одному из нескольких участков кода. Называется также «case structure», в честь одноименной конструкции Pascal. Большинство переключателей в Perl записываются как `for`. См. «Структуры типа Case» в главе 4.

переменная окружения – environment variable

Механизм, посредством которого некий агент более высокого уровня, например, пользователь, может сообщить о своих предпочтениях будущему потомку (дочернему *процессу*, внучатому процессу и т. д.). Каждая переменная окружения представляет собой пару *ключ/значение*, как элемент в *хеше*.

переменная экземпляра – instance variable

Атрибут объекта; данные, хранимые в конкретном объекте, а не в классе как целом.

переменная – variable

Именованный адрес памяти, который может содержать любой из различных типов *значений*, приемлемых для вашей программы.

переносимый – portable

Когда-то относилось к коду C, который можно было компилировать в любой из систем BSD и SysV. В общем, код, который легко преобразовать для выполнения на другой *платформе*, где «легко» можно определить любым способом, что обычно и бывает. Все можно считать переносимым, если хорошенько постараться. См. *передвижной дом* или *Лондонский мост*.

плавающая точка – floating point

Способ хранения чисел в «научной нотации», при котором точность представления числа не зависит от того, насколько оно велико (десятичный разделитель – точка или запятая – «плавают»). Perl производит работу с числами с плавающей точкой, если *целых чисел* оказывается недостаточно. Числа с плавающей точкой являются приближением действительных чисел.

платформа – platform

Аппаратный и программный контексты в целом, в которых выполняется программа. Программа, написанная на языке, зависящем от платформы, может слететь, если изменить машину или операционную систему, или библиотеку, или компилятор, или системную конфигурацию. Интерпретатор *perl* должен отдельно компилироваться для каждой платформы, потому что он реализован на C, но программы, написанные на языке Perl, в значительной мере платформонезависимы.

побочные эффекты – side effects

Нечто дополнительное, что происходит при вычислении *выражения*. В настоящее время это может касаться почти чего угодно. Например, выполнение простой команды присваивания обычно имеет «побочный эффект» присваивания значения переменной. (А вы думали, что присваивание значения было вашей главной целью!) Аналогично присваивание значения специальной переменной `$|` (`$AUTOFLUSH`) имеет побочный эффект, заключающийся в принудительном очищении буфера после каждой операции `write` или `print` с выбранным в данный момент указателем файла.

подкласс – subclass

См. *производный класс*.

подпрограмма – subroutine

Доступный по имени или иным образом участок программы, который можно вызвать из любого места программы, чтобы выполнить некоторую подзадачу программы. Подпрограмма часто имеет параметры, чтобы осуществлять различные, но связанные вещи в зависимости от входных *аргументов*. Если

подпрограмма возвращает осмысленное *значение*, ее называют также *функцией*.

подстановка – substitution

Изменение частей строки через оператор `s///`. (Мы избегаем этого термина в значении *интерполяции переменных*.)

подстрока – substring

Часть *строки*, начинающаяся с некоторой позиции *символа* (*смещения*) и охватывающая некоторое число символов.

подход инструментального ящика – toolbox approach

Идея состоит в том, что с полным набором простых инструментов, хорошо работающих совместно, можно создать почти все, что нужно. Это хорошо, когда надо собрать трехколесный велосипед, но если мы создаем что-то необычное[1], то нам понадобится собственный механический цех, в котором можно создавать специальные инструменты. Perl – своего рода механический цех.

подшаблон кода – code subpattern

Подшаблон *регулярного выражения*, действительной задачей которого является выполнение некоторого кода Perl, например подшаблоны `(?{...})` и `(??{...})`.

подшаблон – subpattern

Компонент шаблона *регулярного выражения*.

позиция косвенного объекта – indirect object slot

Синтаксическая позиция между вызовом метода и его аргументами при использовании синтаксиса вызова косвенного объекта. (Позиция отличается отсутствием запятой между ней и следующим аргументом.) `STDERR` является здесь позицией косвенного объекта:

```
print STDERR "Awake! Awake! Fear, Fire,
    Foes! Awake!\n";
```

поиск по шаблону – pattern matching

Означает взять шаблон, обычно являющийся *регулярным выражением*, и попытаться различными способами наложить его на строку так, чтобы он совпал. Часто применяется для того, чтобы найти в файле интересующие участки данных.

поиск с возвращением – backtracking

Обычай говорить: «Если бы начать все сначала, я бы все сделал иначе» – и затем действительно возвращаться и делать все по-другому. На математическом языке это означает возврат от безуспешной рекурсии по дереву возможностей. Perl осуществляет поиск с возвращением при попытке найти шаблон с помощью *регулярного выражения*, когда из прежних попыток ничего не вышло. См. раздел «Маленькая машинка, которая /(не) может?/» в главе 5 «Поиск по шаблону».

поле – field

Фрагмент числовых или строковых данных, являющийся частью более длинной *строки* или *записи*. Поля переменной ширины обычно разделяются с помощью разделителей (а извлекаются поля посредством `split`), тогда как поля

---

[1] В оригинале говорится о приборе defranishizing comboflux regurgalator – разупрощающем сложнопоточном регургаляторе; слово «*ре***гур***гал*ятор» раньше было длиннее и читалось «регуругалятор», обозначая некое приспособление, при помощи которого йоги могли очень долго вдыхать (inhale) продукты сухой возгонки сложных растительных смесей (в замкнутом цикле, это важно), достигая в результате нирваны и просветления. Впоследствии секрет этого приспособления был утерян, йоги больше не достигают просветления, а лишь впадают в нирвану (внешне это напоминает спячку, только человек при этом сидит и раскачивается, время от времени что-то тихонько напевая). Но слово не исчезло. Буква «у» редуцировалась и термин обозначает теперь довольно несложный, хотя и секретный прибор (регургалятор), состоящий на вооружении армии США. – *Примеч. ред.*

фиксированной ширины обычно находятся в фиксированных позициях (и нужно использовать `unpack`). *Переменные экземпляра* тоже называют полями.

полиморфизм – polymorphism

Идея заключается в том, что можно приказать *объекту* выполнить некоторое родовое действие, и объекты интерпретируют команду различными способами в зависимости от своих типов. [от греческого «много форм»]

полубайт – nybble

Половина *байта*, эквивалентная одной *шестнадцатеричной* цифре и занимающая четыре *бита*.

поразрядный сдвиг – bit shift

Перемещение битов влево или вправо в машинном слове, которое приводит к умножению или делению на степень двух.

порт – port

Часть адреса TCP или UDP, которая направляет пакеты нужному процессу, найдя нужную машину, – нечто вроде добавочного номера, который мы сообщаем, дозвонившись до коммутатора компании. Кроме того, означает результат преобразования кода для платформы, отличной от избранной первоначально.

поступательный поиск – progressive matching

Поиск по шаблону, продолжающий работу с того места, где остановился в предыдущий раз.

постфиксный оператор – postfix

Оператор, который следует за своим *операндом*, например `$x++`.

поток – stream

Поток данных в процесс или из процесса как устойчивая последовательность байтов или символов без видимости разбиения на пакеты. Это своего рода *интерфейс* – соответствующая *реализация* вполне может разбивать данные на пакеты для доставки, но это скрыто от пользователя.

поток – thread

Подобен ответвленному процессу, но без присущей *fork* защиты памяти. Поток меньше по весу, чем полный процесс, т. к. процесс может иметь несколько потоков, выполняемых в нем, которые все борются за пространство памяти одного и того же процесса, если только не предпринимаются шаги для защиты потоков один от другого. См. главу 17 «Потоки».

правое значение – rvalue

Значение, которое можно поместить в правой части *присваивания*. См. также *левое значение*.

прагма (директива) – pragma

Стандартный модуль, практические советы и предложения которого получаются (и, возможно, игнорируются) на этапе компиляции. Имена прагм задаются целиком на нижнем регистре.

предупреждение – warning

Сообщение, выводимое в поток `STDERR` с целью показать, что есть проблемы, но не настолько серьезные, чтобы прекратить работу. См. `warn` в главе 29 и прагму `use warnings` в главе 31 «Модули прагм».

преобразование в строку – stringification

Процесс создания представления абстрактного объекта в виде *строки*.

препроцессинг – preprocessing

То, что некоторый вспомогательный процесс осуществил для преобразования входных данных в форму, более пригодную для текущего процесса. Часто выполняется через входной *канал*. См. также *препроцессор C*.

препроцессор C – C preprocessor

Первый проход типичного компилятора C, в котором обрабатываются строки, начинающиеся символом `#`, для условной компиляции и макроопределений, а также осуществляются различные манипуляции с текстом программы на основе текущих определений. Известен также как *cpp*(1).

префиксный оператор – prefix

Оператор, предшествующий своему *операнду*, например `++$x`.

приведение типа – type casting

Преобразование данных из одного типа в другой. С допускает это. Perl не нуждается в этом. И не хочет этого.

привязка – bind

Назначение конкретного *сетевого адреса сокету*.

приоритет – precedence

Правила поведения, которыми, в отсутствие других указаний, определяется, что должно произойти раньше. Например, при отсутствии скобок умножение всегда выполняется раньше сложения.

присваивание – assignment

Оператор, цель которого – изменять значение *переменной*.

присваивания оператор – assignment operator

Либо обычное *присваивание*, либо составной *оператор*, образованный из обычного присваивания и некоторого другого оператора, который изменяет значение переменной по месту, т. е. относительно ее прежнего значения. Например `$a += 2` прибавляет `2` к `$a`.

пробельный символ – whitespace

Символ, который перемещает курсор, но ничего на экран не выводит. Обычно относится к любому из символов: пробел, табуляция, перевод строки, возврат каретки, подача листа.

программное обеспечение с открытым исходным кодом – open source software

Программы, для которых можно бесплатно получить исходный код и свободно распространять их дальше, без коммерческих условий. Более точное определение можно найти в документе *http://www.opensource.org/osd.html*.

продолжение – continuation

Работа с одной или более физическими строками как с одной логической строкой. Строки *Makefile* продолжаются с помощью обратной косой черты перед *символом перевода строки*. Почтовые заголовки по определению RFC 822 продолжаются с помощью пробела или табуляции *после* перевода строки. В целом для строк Perl не требуется какой-либо знак продолжения, потому что *пробельный символ* (в том числе символ перевода строки) с радостью игнорируется. Обычно.

производный класс – derived class

Класс, который определяет некоторые из своих *методов* через более общий класс, называемый *базовым классом*. Обратите внимание, что классы не разделяются исключительно на базовые или производные: класс может одновременно функционировать как производный и базовый, что, конечно, классно.

пространство имен – namespace

Домен имен. Нет необходимости беспокоиться о том, что имена из одного такого домена используются в другом. См. *пакет*.

протокол – protocol

В сетевом взаимодействии – согласованный способ передачи сообщений в обоих направлениях, который не должен слишком смутить ни одного из корреспондентов.

прототип – prototype

Необязательная часть объявления *подпрограммы*, сообщающая компилятору Perl, сколько и каких аргументов можно передать в качестве *фактических аргументов*, чтобы можно было писать вызовы подпрограмм, которые распознаются анализатором так же, как для встроенных функций. (Или не распознаются, как может случиться.)

процедура – procedure

Подпрограмма.

процесс – process

Экземпляр выполняющейся программы. В многозадачных системах типа Unix два или более отдельных процесса могут одновременно и независимо выполнять

одну и ту же программу – на практике для обеспечения такой счастливой жизни предназначена функция fork. В других операционных системах процессы иногда называются «threads», «tasks», «jobs» (потоки, задачи, задания), часто имея небольшие отличия в значении.

псевдолитерал – pseudoliteral

Оператор, который выглядит похожим на литерал, например оператор захвата выдачи `command`.

псевдоним – alias

Прозвище чего-либо, во всех отношениях ведущее себя так же, как если бы вместо него использовалось исходное имя. Временные псевдонимы неявно создаются в переменной цикла для foreach, в переменной $_ для операторов map или grep, в $a и $b во время выполнения функции сравнения в sort и в каждом элементе @_ для *фактических аргументов* вызова подпрограммы. Постоянные псевдонимы явно создаются в *пакетах* путем *импортирования* символов или присваиваний переменным *typeglob*. Псевдонимы с лексической областью видимости для переменных пакетов явно создаются объявлением our.

псевдофункция – pseudofunction

Конструкция, иногда выглядящая как функция, но в действительности ею не являющаяся. Обычно относится к модификаторам *левых значений*, например my, модификаторам контекста, например scalar, и к конструкциям для выбора собственных кавычек q//, qq//, qx//, qw//, qr//, m//, s///, y/// и tr///.

псевдохеш – pseudohash

Ссылка на массив, первый элемент которого содержит ссылку на хеш. С псевдохешем можно работать и как со ссылкой на массив, и как со ссылкой на хеш.

пустой контекст – void context

Вид *скалярного контекста*, в котором не предполагается возврат *выражением* какого-либо *значения*, а вычисление его осуществляется только ради *побочных эффектов*.

путь, маршрут – pathname

Полностью квалифицированное имя файла, например */usr/bin/perl*. Иногда смешивается с PATH.

рабочий каталог – working directory

Текущий *каталог*, от которого операционная система ведет относительные маршруты. *Операционная система* определяет текущий каталог как тот, который был указан ей посредством команды chdir, или как то место, в котором находился ваш родительский *процесс* при вашем появлении на свет.

разбиение на лексемы – tokenizing

Расщепление текста программы на *лексемы*. Другое название – «lexing».

разделитель – separator

Символ или *строка*, позволяющие разграничить две окружающие их строки. С разделителями работает функция split. Не путать с *ограничителями* и *терминаторами*. Союз «и» в предыдущем предложении был разделителем двух альтернатив.

разрушать – destroy

Освобождать память *объекта ссылки* (сначала вызывая метод DESTROY, если он есть).

разыменовывать – dereference

Причудливый термин вычислительной техники, означающий «следовать за *ссылкой* к тому, на что она указывает». Часть «раз» относится к тому обстоятельству, что вы удаляете один уровень *косвенности*.

разыменовывающий символ – funny character

Некто, похожий на Ларри или одного из его эксцентричных друзей (funny character – это и «забавная фигура»). Относится также к необычным префиксам, которыми Perl требует помечать свои переменные-существительные.

расширение – extension

Модуль Perl, который загружает также скомпилированный код C или C++. В более широком смысле – любая экс-

периментальная особенность, которая вставляется в Perl в результате компиляции, например многозадачность.

реализация – implementation

То, как фрагмент кода фактически выполняет свою задачу. Пользователи кода не должны полагаться на то, что детали реализации остаются постоянными, если только они не входят в опубликованный *интерфейс*.

регулярное выражение – regular expression

Объект, который имеет несколько интерпретаций, как слон. Для ученого в области вычислительных наук это грамматика небольшого языка, в котором одни строки допустимы, а другие – нет. Для обычных людей это шаблон, который можно использовать, чтобы найти то что нужно, если оно иногда меняется. Регулярные выражения Perl далеки от регулярности в теоретическом смысле, но при регулярном применении они работают вполне хорошо. Вот регулярное выражение: `/Oh s.*t./`. Оно соответствует таким строкам, как `"Oh say can you see by the dawn's early light"` и `"Oh sit! "`. См. главу 5.

регулярный файл – regular file

Файл, который не является *каталогом*, *устройством*, именованным *каналом*, *сокетом* или *символической ссылкой*. Perl идентифицирует регулярные файлы с помощью оператора проверки файла `-f`. Иногда они называются «плоскими» (plain) файлами.

режим – mode

В контексте системного вызова *stat*(2) указывает на поле, содержащее *биты разрешений* и тип *файла*.

рекурсия – recursion

Искусство определить что-либо (хотя бы частично), ссылаясь на самое себя. В словарях это порочная практика, но в компьютерных программах часто работает хорошо, если следить за тем, чтобы не образовывалась бесконечная рекурсия, которая сходна с бесконечным циклом, но влечет более зрелищный отказ.

родительский класс – parent class

См. *базовый класс*.

самое левое самое длинное – leftmost longest

Стремление механизма *регулярных выражений* найти самое левое соответствие *шаблона*; затем, исходя из позиции, в которой найдено соответствие, стремление найти самое длинное соответствие (в предположении использования *жадного* квантификатора). Подробности об этом предмете см. в главе 5.

самооживление – autovivification

Греко-романское слово (autovivication, конечно). В Perl адреса хранения (*левые значения*) самопроизвольно создаются по мере необходимости, в том числе создаются значения *жестких ссылок*, указывающих на следующий уровень хранения. Присваивание `$a[5][5][5][5][5] = "quintet"` потенциально создает пять адресов хранения скаляров плюс четыре ссылки (в первых четырех адресах скаляров), указывающие на четыре новых анонимных массива (для хранения четырех последних адресов скаляров). Но смысл самооживления в том, что программисту не надо об этом беспокоиться.

сборка мусора – garbage collection

Функция, которой дано неправильное имя, – ее следовало назвать «расчет на то, что мать уберет после вас». Строго говоря, Perl этого не делает, а полагается на механизм подсчета ссылок, который приведет все в порядок. Однако мы редко выражаемся строго, и будем часто называть схему подсчета ссылок формой уборки мусора. (Может быть, вас утешит, что по завершении работы интерпретатора запускается «настоящий» сборщик мусора, который обеспечит уборку, если вы неряшливо обращались с циклическими ссылками и тому подобным.)

свойство символа – character property

Предопределенный *класс символов*, соответствующих *метазнаку* `\p`. Для

Unicode определено много стандартных свойств символов.

свойство – property

См. *переменная экземпляра* или *свойство символа*.

связка – bundle

Группа взаимосвязанных модулей в *CPAN*. (Иногда также означает группу ключей командной строки, объединенных в один *кластер ключей*.)

связующий язык – glue language

Язык, например, Perl, который удобен для соединения вместе вещей, изначально для соединения не предназначенных.

связь – tie

Связь между магической переменной и классом ее реализации. См. функцию `tie` в главе 29 и главу 14 «Связанные переменные».

сдвиг влево – left shift

Поразрядный сдвиг, который умножает число на некоторую степень 2.

сдвиг вправо – right shift

Поразрядный сдвиг, который делит число на некоторую степень 2.

семафор – semaphore

Вид взаимной блокировки, предотвращающий одновременное использование одних и тех же ресурсов несколькими *потоками* или *процессами*.

сервер – server

В сетевом взаимодействии – *процесс*, который рекламирует *сервис* или просто ждет в известном месте *клиентов*, которые обращаются к нему за сервисом.

сервис – service

Нечто, что мы выполняем для кого-то другого, чтобы удовлетворить его, например, сообщаем ему время суток (или время, которое ему осталось). На некоторых машинах известные сервисы могут быть перечислены функцией `getservent`.

сериализация – serialization

Расположение сложной *структуры данных* в линейном порядке, чтобы ее можно было сохранить в виде *строки* в дисковом файле или базе данных либо отправить в *канал*. Другое название – пересылка (marshalling).

сетевой адрес – network address

Важнейший атрибут сокета, подобный телефонному номеру. Обычно IP-адрес. См. также *порт*.

сигнал – signal

Гром среди ясного неба; событие, запущенное *операционной системой*, когда вы меньше всего его ожидали.

символ – character

Небольшое целое число, представляющее орфографическую единицу. Исторически символы хранились как целые числа фиксированной ширины (обычно в байте или двух, в зависимости от набора символов), но с появлением UTF-8 символы часто стали хранить в переменном числе байтов в зависимости от размера целого числа, представляющего символ. Perl, по большей части, управляется с этим прозрачным для пользователя образом.

символ – symbol

В целом любая *лексема* или *метазнак*. Часто употребляется в более узком смысле для обозначения имени, которое можно найти в *таблице символов*.

символическая ссылка – symbolic link

Альтернативное имя файла, указывающее на действительное имя файла, которое, в свою очередь, указывает на действительный *файл*. Когда *операционная система* пытается анализировать *маршрут*, содержащий символическую ссылку, она просто подставляет новое имя и продолжает анализ.

символическая ссылка – symbolic reference

Переменная, значением которой является имя другой переменной или под-

программы. Путем *разыменования* первой переменной можно получить вторую. Символические ссылки недопустимы при использовании директивы `use strict 'refs'`.

символический отладчик – symbolic debugger

Программа, которая позволяет выполнять программу пошагово, время от времени останавливаясь и выводя данные, чтобы посмотреть, не пошло ли что-нибудь не так, и если да, то что. Слово «символический» означает, что можно разговаривать с отладчиком, используя те же символы, с которыми была написана ваша программа.

синтаксис – syntax

От греческого. Порядок, в соответствии с которым вещи (в особенности, символы) располагаются совместно.

синтаксический анализ – parsing

Тонкое, но иногда жестокое искусство попытки преобразования вашей, возможно, плохо сформированной программы в допустимое *синтаксическое дерево*.

синтаксическое дерево – syntax tree

Внутреннее представление программы, при котором конструкции более низкого уровня висят из заключающих их конструкций более высокого уровня.

синтаксическое подслащение – syntactic sugar

Альтернативный способ более простого написания; сокращение.

синхронный – synchronous

Программирование, при котором можно определить организованную последовательность событий; т. е. когда события происходят одно за другим, а не в одно и то же время.

системный вызов – syscall

Вызов *функции*, непосредственно обращенный к *операционной системе*. Многие важные подпрограммы и функции, которые мы используем, не являются непосредственными системными вызовами, но строятся на один или более уровней выше, чем системные вызовы. В целом, программистам Perl не нужно беспокоиться о различии. Однако тот, кто будет знать, какие из функций Perl в действительности являются системными вызовами, сможет предсказать, какая из них установит переменную `$!` (`$ERRNO`) в случае отказа. К сожалению, начинающие программисты часто ошибочно применяют термин «системный вызов» к вызову функции Perl `system`, при котором фактически выполняется много системных вызовов. Чтобы избежать путаницы, мы почти всегда говорим «системный вызов», когда имеется в виду то, что можно косвенно вызвать через функцию Perl `syscall`, и никогда в отношении того, что следует вызывать функцией Perl `system`.

скаляр – scalar

Простое отдельное значение; число, *строка* или *ссылка*.

скалярная переменная – scalar variable

Переменная с префиксом `$`, содержащая одно значение.

скалярное значение – scalar value

Значение, являющееся скаляром, в отличие от *списка*.

скалярный контекст – scalar context

Ситуация, в которой окружение (вызывающий код) ожидает от *выражения*, что оно возвратит одно *значение*, а не *список* значений. См. также *контекст* и *списковый контекст*. Скалярный контекст иногда налагает дополнительные ограничения на возвращаемое значение – см. *строковый контекст* и *числовой контекст*. Иногда мы говорим о *булевом контексте* внутри условных операторов, но он не налагает дополнительных ограничений, потому что любое скалярное значение, число или *строка* уже является истинным или ложным.

скалярный литерал – scalar literal

Число или строка в кавычках – фактическое *значение* в вашей программе, в отличие от *переменной*.

слово – word

В обычном компьютерном языке фрагмент данных такого размера, который наиболее эффективно обрабатывается компьютером, обычно 32 бита или отличное от него в ту или другую сторону на какую-то степень двух. В культуре Perl чаще относится к буквенно-цифровому *идентификатору* (включая символы подчеркивания) или к строке *символов*, отличных от пробельных, ограниченной по краям пробельными символами или границей строки.

смертник – mortal

Временное значение, которое должно умереть, когда закончится текущая команда.

смещение – offset

Количество элементов, которое следует пропустить, перемещаясь от начала строки или массива в заданную позицию. Поэтому минимальное смещение равно нулю, а не единице, поскольку не надо ничего пропускать, чтобы добраться до первого элемента.

соединение – connection

В телефонии это создание временной электрической цепи между аппаратами вызвавшего и вызываемого абонентов. В сетевом взаимодействии это такая же временная цепь между *клиентом* и *сервером*.

сокет – socket

Конечная точка при связи по сети между несколькими *процессами*, которая действует во многом сходно с телефонной кабинкой или почтовым ящиком. Самое важное в сокете – его *сетевой адрес* (как номер телефона). Различные виды сокетов имеют различные виды адресов: некоторые похожи на телефонные номера, а некоторые – нет.

соответствие – matching

См. *поиск по шаблону*.

сортирующая последовательность – collating sequence

Порядок, согласно которому сортируются *символы*. Им руководствуются, например, программы сортировки *строк*, решая, в какое место данного глоссария поместить «порядок сравнения».

списковое значение – list value

Безымянный список временных скалярных величин, который может передаваться в программе от любой функции, генерирующей список, любой функции или конструкции, предоставляющей *списковый контекст*.

списковый контекст – list context

Ситуация, в которой окружение (вызывающий код) ожидает от *выражения*, что оно возвратит список значений, а не отдельное значение. Функции, которым требуется список аргументов, сообщают этим аргументам, что они должны создать списковое значение. См. также *контекст*.

списковый оператор – list operator

Оператор, который что-то делает со списком значений, например `join` или `grep`. Обычно используется для именованных встроенных операторов (таких, как `print`, `unlink` и `system`), в которых круглые скобки вокруг списка *аргументов* не нужны.

СПИСОК – LIST

Синтаксическая конструкция, представляющая разделенный запятыми список выражений, производящих при вычислении *списковое значение*. Каждое *выражение* в `LIST` вычисляется в *списковом контексте* и интерполируется в списковое значение.

список – list

Упорядоченное множество скалярных значений.

срез – slice

Произвольное число элементов, отобранное из списка, массива или хеша.

ссылка на найденный текст – backreference

Подстрока, *захваченная* подшаблоном в простых скобках в *регулярном выражении*. Десятичные числа с обратной косой чертой перед ними (`\1`, `\2` и т. д.) да-

лее в том же шаблоне ссылаются на соответствующий подшаблон в текущем поиске. Вне шаблона нумерованные переменные (`$1`, `$2` и т. д.) продолжают ссылаться на те же значения, пока этот шаблон является последним успешно найденным шаблоном в текущей динамической области видимости.

ссылка – reference

Место, в котором мы ищем указатель на информацию, находящуюся где-то еще. (См. *косвенность*.) Есть два рода ссылок: *символические ссылки* и *жесткие ссылки*.

ссылка; компоновать – link

При использовании в качестве существительного – имя в *каталоге*, представляющее *файл*. На заданный файл может быть несколько ссылок. Это похоже на включение в телефонный справочник одного и того же номера телефона под различными именами. Как глагол означает разрешение неразрешенных символов частично скомпилированного файла в (почти) исполняемый образ. Компоновка обычно может быть статической или динамической, что не имеет никакого отношения к статической или динамической области видимости.

стандартное устройство ввода – standard input

Входной *поток* программы по умолчанию, который, если это возможно, не должен обращать внимания на то, откуда поступают данные. Представлен в программе Perl *указателем файла* `STDIN`.

стандартное устройство вывода – standard output

Выходной *поток* программы по умолчанию, который, если это возможно, не должен обращать внимания на то, куда отправляются данные. Представлен в программе Perl *указателем файла* `STDOUT`.

стандартное устройство ошибок – standard error

Стандартный выходной *поток* для неприятных замечаний, которые не принадлежат *стандартному устройству вывода*. В программе Perl представлен *указателем файла* `STDERR`. Можно явно использовать этот поток, а встроенные функции `die` и `warn` пишут в стандартное устройство вывода автоматически.

стандартный ввод/вывод – standard I/O

Стандартная библиотека C для осуществления *буферизованного* ввода и вывода в *операционную систему*. («Стандарт» стандартного ввода/вывода лишь частично связан со «стандартами» стандартных устройств ввода и вывода.) В целом Perl полагается на реализацию стандартного ввода/вывода, предоставляемую операционной системой, поэтому характеристики буферизации программы Perl на одной машине могут не вполне совпадать с подобными характеристиками на другой машине. Обычно это влияет только на эффективность, а не на семантику. Если ваш пакет стандартного ввода/вывода осуществляет только блочную буферизацию, а вам нужно *очищать буфер* чаще, установите переменную `$|` равной истинному значению.

стандартный – standard

Включенный в официальный дистрибутив Perl – стандартный модуль, стандартное инструментальное средство, стандартная страница руководства Perl.

статическая область видимости – static scoping

Нет такой вещи. См. *лексическая область видимости*.

статическая переменная – static variable

Нет такой вещи. Используйте *лексическую переменную* в области видимости, большей чем ваша *подпрограмма*.

статический метод – static method

Нет такой вещи. См. *метод класса*.

статический – static

Медленно меняющийся относительно чего-то другого. (К несчастью, все является относительно стабильным в сравнении с чем-то еще, за исключением неко-

торых элементарных частиц, в отношении которых нельзя быть уверенным.) В компьютерах, где все должно изменяться быстро, «статический» имеет презрительный оттенок, указывая на не вполне функциональные *переменную*, *подпрограмму* или *метод*. В культуре Perl этого слова вежливо избегают.

статус завершения – exit status

См. *статус*.

статус – status

Значение, возвращаемое родительскому *процессу*, когда один из его дочерних процессов умирает. Это значение помещается в специальную переменную `$?`. Ее верхние восемь разрядов являются статусом завершения покойного процесса, а нижние восемь указывают сигнал (если он был), от которого процесс умер. В системах Unix это значение статуса то же, что и слово статуса, возвращаемое *wait*(2). См. `system` в главе 29.

стек – stack

Устройство, на вершину которого можно что-то помещать, а затем снимать в обратном порядке. См. *LIFO*.

страница руководства – manpage

«Страница» из руководства, доступ к которой обычно осуществляется через команду *man*(1). Страница руководства содержит SYNOPSIS, DESCRIPTION, список BUGS и т. д., что обычно длиннее страницы. Есть страницы руководства, документирующие команды, системные вызовы, библиотечные функции, устройства, протоколы, файлы и тому подобное. В данной книге мы называем страницей руководства любую часть стандартной документации Perl (например, *perlop* или *perldelta*), независимо от того, в каком формате она установлена в конкретной системе.

строка – string

Последовательность символов, такая как «He said !@#*&%@#*?!». Строка не обязательно должна быть полностью печатаемой.

строковый контекст – string context

Ситуация, в которой окружение (вызывающий код) ожидает от *выражения*, что оно возвратит строку. См. также *контекст* и *числовой контекст*.

строчка – line

В Unix это последовательность из нуля или более символов, отличных от перевода строки, завершающаяся символом *перевода строки*. На не-Unix-машинах они эмулируются библиотекой C, даже если *операционная система* имеет иные представления.

структура – structure

См. *структура данных*.

структура stat – stat structure

Особое место внутри Perl, где хранятся данные о последнем *файле*, сведения о котором запрашивались.

структура данных – data structure

Способ, которым отдельные элементы данных связаны друг с другом, и форма, которую они образуют все вместе – прямоугольная таблица или треугольное дерево.

суперпользователь – superuser

Лицо, которому операционная система позволяет делать почти все. Обычно это ваш системный администратор или некто, претендующий на роль системного администратора. В системах Unix это пользователь *root*. В системах Windows это обычно пользователь Administrator.

сценарий – script

Текстовый *файл*, являющийся программой, предназначенной для непосредственного *исполнения*, а не *компиляции* в файл другого вида перед исполнением. Кроме того, в контексте *Unicode* означает систему записи для некоторого языка или группы языков, например греческого, бенгальского или Klingon (см. главу 15).

таблица символов – symbol table

Место, где *компилятор* запоминает символы. Программа типа Perl должна

каким-то образом запоминать все имена всех *переменных*, *указателей файлов* и *подпрограмм*, которые были использованы. Она делает это, помещая имена в таблицу символов, которая реализована Perl с помощью *хеш-таблицы*. В каждом пакете есть своя таблица символов, предоставляющая каждому *пакету* собственное *пространство имен*.

текст – text

Строка или *файл*, содержащие, в основном, выводимые символы.

текущий выбранный канал вывода – currently selected output channel

Последний *указатель файла*, который был назначен с помощью `select(FILEHANDLE)`; `STDOUT`, если не был выбран никакой указатель файла.

текущий пакет – current package

Пакет, в котором скомпилирована текущая команда. Ищите обратно по тексту вашей программы в текущей *лексической области видимости* или в охватывающих лексических областях, пока не найдете объявление пакета. Это и будет именем текущего пакета.

текущий рабочий каталог – current working directory

См. *рабочий каталог*.

терм – term

От «терминальный», т. е. лист *синтаксического дерева*. То, что грамматически функционирует как *операнд* для операторов в выражении.

терминатор – terminator

Символ или *строка*, помечающие конец другой строки. Переменная `$/` содержит строку, которая завершает операцию `readline`, которую `chomp` удаляет из конца строки. Не путать с *ограничителями* и *разделителями*. Точка в конце этого предложения является терминатором.

тернарный – ternary

Оператор, принимающий три *операнда*. Иногда произносится *trinary*.

тест пустого подкласса – empty subclass test

Означает, что пустой *производный класс* должен вести себя, в точности как его *базовый класс*.

тип данных – data type

Множество допустимых значений, а также операции, умеющие обращаться с этими значениями. Например, для числового типа данных есть некое множество чисел, с которыми можно работать, а также различные математические операции, которые можно производить с числами, но которые не очень осмысленны, скажем, со строками. Для строк есть свои операции, например *конкатенация*. А для составных типов, образованных из некоторого количества меньших частей, обычно существуют операции образования и декомпозиции, а возможно, и реорганизации. *Объекты*, моделирующие предметы реального мира, часто имеют операции, соответствующие их реальной деятельности. Например, если моделируется лифт, то объект лифта должен иметь *метод* `открыть_дверь ()`.

тип – type

См. *тип данных* и *класс*.

типизованная лексическая переменная – typed lexical

Лексическая переменная, объявленная с типом класса: `my Pony $bill`.

транслитерировать – transliterate

Преобразовывать одно представление строки в другое путем отображения каждого символа исходной строки в соответствующий символ результирующей строки. См. оператор `tr///` в главе 5.

триггер – trigger

Событие, которое вызывает запуск обработчика.

троичный – trinary

Не система из трех звезд, а оператор, принимающий три операнда. Иногда пишется *ternary*.

тупоконечник – big-endian

Из Свифта: тот, кто разбивает яйцо с тупого конца. Употребляется также в отношении компьютеров, в которых адрес хранения более значащего *байта* слова младше, чем адрес менее значащего байта. Часто такие компьютеры считаются превосходящими по сравнению с «остроконечными» машинами. См. также *остроконечник*.

тыква – pumpkin

Условный «жезл», ходящий в сообществе Perl и указывающий на ведущий интегратор в какой-либо области разработок.

уборка – reaping

Последний ритуал, исполняемый родительским *процессом* в отношении почившего дочернего процесса, чтобы тот не остался *зомби*. См. вызовы функций `wait` и `waitpid`.

указатель каталога – directory handle

Имя, представляющее конкретный случай открытого каталога для чтения его вплоть до закрытия. См. функцию `opendir`.

указатель файла – filehandle

Идентификатор (не обязательно связанный с действительным именем файла), представляющий конкретный экземпляр открытия файла, пока он не будет закрыт. Если предполагается открывать и закрывать несколько разных файлов подряд, то удобно открывать их все через один и тот же указатель файла, а не писать отдельный код для обработки каждого файла.

указатель – pointer

Переменная в языке типа C, содержащая точный адрес в памяти для другого элемента. Perl обрабатывает указатели внутренне, и о них беспокоиться не надо. Вместо них используются символические указатели в виде *ключей* и имен *переменных* либо *жесткие ссылки*, которые указателями не являются (но действуют как указатели и фактически содержат в себе указатели).

унарный оператор – unary operator

Оператор только с одним *операндом*, такой как `!` или `chdir`. Унарные операторы обычно являются префиксными, т. е. они предшествуют своему операнду. Операторы `++` и `--` могут быть префиксными или постфиксными. (Их положение *изменяет* их смысл.)

усечение – truncating

Удаление из файла его содержания – автоматически при открытии файла для записи или явно с помощью функции `truncate`.

условный оператор – conditional

Нечто «iffy» (сомнительное). См. *булев контекст*.

устройство – device

Аппаратная штуковина (типа диска, ленточного устройства, джойстика или мыши), прикрепленная к вашему компьютеру, которую *операционная система* пытается заставить выглядеть как *файл* (или группа файлов). В Unix эти ложные файлы обычно живут в каталоге */dev*.

утверждение – assertion

Составляющая *регулярного выражения*, которая должна быть истинной в случае соответствия шаблону, но не обязательно должна сама соответствовать каким-то символам. Часто используется узко в значении утверждения *нулевой ширины*.

фаза исполнения – run phase

Любой момент времени после того, как Perl запустит нашу основную программу. См. также *фаза компиляции*. Фаза исполнения в основном проходит в *этапе* (или времени) *исполнения* (run time), но может также проходить в *этапе* (или времени) *компиляции* (compile time), когда выполняются операторы `require`, `do` *`FILE`* или `eval` *`STRING`* либо когда подстановка использует модификатор `/ee`.

фаза компиляции – compile phase

Любой момент времени перед тем, как Perl начинает выполнять вашу основную программу. См. также *фаза испол-*

нения. Фаза компиляции в основном приходится на *этап компиляции*, но может также иметь место на *этапе исполнения*, когда вычисляются блоки BEGIN, объявления use или подвыражения констант. Код начального запуска и импорта любого объявления use также выполняется в фазе компиляции.

файл – file

Именованная совокупность данных, обычно хранящаяся на диске в *каталоге файловой системы*. Примерно соответствует документу, если использовать канцелярские метафоры. В современных файловых системах можно давать файлам несколько имен. Некоторые файлы обладают специальными свойствами, например, каталоги и устройства.

файловая система – filesystem

Набор *каталогов* и *файлов*, находящихся в разделе диска. Иногда называется «разделом» (partition) диска. Можно изменять имена файлов и даже перемещать файлы из каталога в каталог в пределах файловой системы без фактического перемещения самого файла, по крайней мере, в Unix.

фактические аргументы – actual arguments

Скалярные значения, передаваемые *функции* или *подпрограмме* при их вызове. Например, при вызове power("puff") строка "puff" представляет собой фактический аргумент. См. также *аргумент* и *формальные параметры*.

фатальная ошибка – fatal error

Не перехваченная *исключительная ситуация*, вызывающая завершение *процесса* после вывода сообщения в стандартный поток ошибок. Ошибки, происходящие внутри eval, не являются фатальными. Завершение eval происходит после того, как сообщение об исключительной ситуации помещается в переменную $@ ($EVAL_ERROR). Можно попытаться спровоцировать фатальную ошибку посредством оператора die (который возбуждает исключительную ситуацию), но она может быть перехвачена динамически охватывающим eval. В отсутствие перехвата die становится фатальной ошибкой.

фильтр исходного кода – source filter

Модуль особого типа, осуществляющий *препроцессинг* нашего сценария перед тем, как он попадет к *лексическому анализатору*.

фильтр – filter

Программа, предназначенная для того, чтобы преобразовать входной *поток* в выходной поток.

флаг – flag

Мы стараемся избегать этого термина ввиду его многозначности. Он может означать *ключ* командной строки, который сам не принимает аргументов (например, флаги Perl -n и -p) или, реже, индикаторный разряд (например, флаги O_CREAT и O_EXCL в sysopen).

формальные аргументы – formal arguments

Общие имена, под которыми *подпрограмме* известны ее *аргументы*. Во многих языках формальным аргументом всегда даются личные имена, но в Perl формальные аргументы просто являются элементами массива. Формальными аргументами в программе Perl являются $ARGV[0], $ARGV[1] и т. д. Аналогично формальными аргументами в подпрограмме Perl являются $_[0], $_[1] и т. д. Аргументам можно дать отдельные имена, присвоив их значения списку my. См. также *фактические аргументы*.

формат – format

Указывает, сколько куда помещать пробелов, цифр и прочего, чтобы все, что вы выводите, выглядело аккуратно и красиво.

формирователь – composer

«Конструктор» для *объекта ссылки*, который в действительности не является *объектом*, например, анонимного массива или хеша. Например, пара фигурных скобок действует как формирователь хеша, а пара квадратных скобок действует как формирователь для мас-

сива. См. раздел «Создание ссылок» в главе 8 «Ссылки».

функция – function

В терминах математики – отображение каждого набора входных значений в некоторое выходное значение. В компьютерах указывает на *подпрограмму* или *оператор*, возвращающие *значение*. Она может и не иметь входных значений (называемых *аргументами*).

хакер – hacker

Некто, проявляющий поразительную настойчивость в решении технических проблем, будь то игра в гольф, сражение с орками или программирование. Хакер является нейтральным термином в смысле морали. Добрых хакеров не нужно путать со злыми *крэкерами* или невежественными *script kiddies*. Если вы их путаете, то мы должны предположить, что вы злы или невежественны.

хеш – hash

Неупорядоченное объединение пар *ключ/значение*, хранение которых позволяет легко использовать строку *ключ* для поиска связанного с ней *значения* данных. Этот глоссарий подобен хешу, в котором определяемое слово служит ключом, а определение является значением. Иногда хеши называют «ассоциативным массивом», длина которого служит достаточным основанием предпочесть слово «хеш».

хеш-таблица – hash table

Структура данных, внутренне используемая Perl для эффективной реализации ассоциативных массивов (хешей). См. также *блок – bucket*.

хост – host

Компьютер, на котором располагается программа или другие данные.

целое число – integer

Число без дробной (десятичной) части. Числа по порядку, например, 1, 2, 3 и т. д., но вместе с 0 и отрицательными числами.

цикл – loop

Конструкция, которая выполняет что-то многократно, как американские горы.

ЧАВО – FAQ

Frequently Asked Question – часто задаваемые вопросы (не обязательно с часто получаемыми ответами, особенно если ответ есть в Perl FAQ, входящем в стандартную поставку Perl).

числовой контекст – numeric context

Ситуация, в которой окружение (вызывающий код) предполагает, что выражение возвращает число. См. также *контекст* и *строковый контекст*.

читаемый – readable

В применении к файлу означает, что установлен бит разрешений, позволяющий обращаться к файлу. В применении к компьютерной программе означает, что она написана достаточно хорошо, чтобы можно было разобраться в том, что она должна делать.

члены-данные – member data

См. *переменная экземпляра*.

шаблон этапа исполнения – run-time pattern

Шаблон, содержащий одну или более переменных, которые должны быть интерполированы прежде разбора шаблона как *регулярного выражения*, который поэтому нельзя анализировать на этапе компиляции, но следует заново анализировать при каждом выполнении оператора поиска по шаблону. Шаблоны этапа исполнения полезны, но дороги.

шаблон – pattern

Структура, используемая при *поиске по шаблону*.

шестнадцатеричный – hexadecimal

Число по основанию 16, «hex». Числа от 10 до 16 обычно представляются буквами от `a` до `f`. Шестнадцатеричные константы в Perl начинаются с `0x`. См. также функцию `hex` в главе 29.

широковещание – broadcast

Отправка *датаграммы* одновременно в несколько мест.

экземпляр – instance

Сокращенно от «экземпляр класса», что означает *объект класса*.

эклектический – eclectic

Порожденный многими источниками. Некоторые скажут *слишком* многими.

экспорт – export

Сделать символы *модуля* доступными для *импорта* другими модулями.

элемент – element

Основной строительный элемент. Если говорить о массиве, то это один из предметов, из которых он составлен.

этап исполнения – run time

Время, когда Perl фактически выполняет то, что сказано в нашем коде, в противоположность предшествующему периоду, когда он пытался выяснить, есть ли вообще смысл в том, что мы сказали, и который был *этапом компиляции*.

этап компиляции – compile time

Этап, на котором Perl пытается осмыслить ваш код, в противоположность тому времени, когда ему кажется, что он знает, каков смысл вашего кода, и просто пытается сделать то, что, по его мнению, ваш код хочет, чтобы было сделано, и это *этап исполнения*.

является – is-a

Связь между двумя объектами, при которой один объект рассматривается как более специфическая версия другого, родового объекта: «верблюд относится к млекопитающим». Поскольку родовой объект в действительности существует только в платоническом смысле, мы обычно добавляем долю абстракции к понятию объектов и представляем связь как существующую между родовым *базовым классом* и специфическим *производным классом*. Как ни странно, у платонических классов связи бывают не только платоническими – см. *наследование*.

Алфавитный указатель

Специальные символы

( ) (круглые скобки), 43, 125, 126, 641, 659
(...) группировка, 201
(?:PATTERN) обозначение, кластеризация без захвата, 228
автоматическая вставка в Deparse, 532
в вызовах подпрограмм, 264, 278
в качестве символов кавычек, 111
в списочных значениях, 108
в ссылках на найденный текст, 74, 226
в функциях, 737
для захвата в шаблонах, 182, 187, 194
метасимвол, 181, 200
нейтрализация, 259
нулевые списки, представление, 110
условный оператор, использование с, 143
[] (квадратные скобки)
в классах символов, 201, 204, 208
в массивах, 40
для Unicode и, 457, 458
как метазнаки, 181, 200
приоритет, 126
формирователь массива, 292
{ } (фигурные скобки), 311, 641, 658
{ метасимвол, 181, 200
в блоках команд, 63, 84
в командах отладчика, 566
в строках, 338
в форматах, 281
голые блоки, создание, 162
для Unicode, 206, 456
для блоков операторов, 151
для хешей в отличие от массивов, 44
идентификаторы в, 98
квантификатор, 71, 201
охватывающие ключи хешей, 42
поиск файлов по шаблону в старину и, 121
приоритет, 126
стрелки и, 301
формирователь хешей, 292
шаблоны поиска, использование для устранения неоднозначности, 101
& (амперсанд), 134
&& (логическое И) оператор, 59, 140, 362
& (поразрядное И) оператор, 139
&&= (присваивание логического И) оператор, 144
&= (присваивание поразрядного И) оператор, 144
Perl версии 5, изменения в использовании, 48
в именах подпрограмм, 37, 88, 89, 166, 264
в именах файлов, 813
оператор взятия адреса (в C), 148
символ прототипа, 272
| (вертикальная черта)
|| команда отладчика, 567
-| конвейера псевдокоманда, 480
|- конвейера псевдокоманда, 479
|| (логическое ИЛИ) оператор, 140, 233, 362
приоритет относительно chdir, 126
| (поразрядное ИЛИ) оператор, 139
||= (присваивание логического ИЛИ) оператор, 144
|= (присваивание поразрядного ИЛИ) оператор, 144
...|... чередование, 182, 201, 230
в именах файлов, 810
для выравнивания по центру, 285

команда отладчика, 567
метасимвол, 181, 200
? (вопросительный знак), 134
? квантификатор, 71, 201, 219
?? квантификатор, 202, 219
?: (условный) оператор, 142, 164
метасимвол, 181, 200
? (вопросительный знак) расширения регулярных выражений
(?<=), 249
(?!), 248
(?=), 248
(?>), 249
! (восклицательный знак)
! (в командах отладчика), 566
!! команда отладчика, 567
! (логическое отрицание) оператор, 129
перегрузка, 401
!= (не равно) оператор, 60, 138, 640
!~ (связывания) оператор, 130
использование с операторами поиска по шаблону, 184
дополнение наборов символов, использование для, 217
- (дефис), 134, 246
- - (декремента) оператор, 58, 128
- - ключ командной строки, 543
-*- для emacs, 539
-= (присваивание вычитания) оператор, 144
-> (стрелка) оператор, 127, 299
вызов метода и, 359
разыменование ссылок, 127
арифметического отрицания оператор, 129
в диапазонах символов, 208
в командах отладчика, 565
вычитание наборов символов, 216
вычитания оператор, 132
(диез)
#! (shebang) нотация, 50, 538
проблемы защиты данных при использовании, 623
моделирование на не-Unix системах, 540
в комментариях, 84
/x модификатор шаблона, использование с, 189
в форматах, 281
как символ кавычек, пробелы и, 100
, (запятая), 42
большие числа и, 95
в командах print, 729
использование в объявлениях циклов, 155
как разделитель, 43
оператор, 146
скалярный контекст, 647
ошибка в командах print, 640
пары ключ/значение и, 42, 113
приоритет, 126, 170
разделитель значений в списке, 108, 111
* (звездочка), 134
** (возведение в степень) оператор, 129
*? квантификатор, 201
**= (присваивание возведения в степень) оператор, 144
*= (присваивание умножения) оператор, 144
в именах typeglob, 37, 89, 115
для поиска имен по шаблону, 121
квантификатор, 71, 201, 219
метасимвол, 181, 200
мультипликативный оператор, 131
оператор умножения, 126
разадресации оператор (в C), 148
символ прототипа, 272
@ (знак at)
@_ (@ARG) массив, 265, 717
@+ (@LAST_MATCH_END) массив конечных позиций, 228, 726
@– (@LAST_MATCH_START) массив начальных позиций, 228, 726
в именах массивов, 36, 88
в отладчике, 559
в строках шаблонов, 281
внутри строк, преобразование с помощью обратной косой черты, 97
изменения между версиями Perl, 645
пример использования, 892
символ прототипа, 272
$ (знак доллара), 42
$< ($REAL_USER_ID), 732
$;($SUBSCRIPT_ SEPARATOR), 115, 734
$#
префикс для последнего индекса массива, 113

устаревшая переменная для числового вывода, 716
$*, 716
$_ ($ARG) переменная, 69, 72, 716
glob функция и, 121
grep функция и, 791
map функция и, 801
в операторах foreach, 156
оператор угловой скобки и, 117
$? ($CHILD_ERROR), 90, 718
close функция и, 753
оператор обратные кавычки и, 117
$) ($EFFECTIVE_GROUP_ID), 719
$> ($EFFECTIVE_USER_ID), 719
$! ($ERRNO, $OS_ERROR), 60, 683
$@ ($EVAL_ERROR), 720
$- ($FORMAT_LINES_LEFT), 283, 286, 722
$= ($FORMAT_LINES_PER_PAGE), 284, 722
$~ ($FORMAT_NAME), 284, 641, 722
$% ($FORMAT_PAGE_NUMBER), 284, 722
$^ ($FORMAT_TOP_NAME), 284, 286, 641, 723
$. ($INPUT_LINE_NUMBER), 141
сброс с помощью функции close, 753
$/ ($INPUT_RECORD_SEPARATOR), 543, 724
$+ ($LAST_PAREN_MATCH), 228, 727
$& ($MATCH), 187, 228, 649, 728
$| ($OUTPUT_AUTOFLUSH), 284, 641, 729
$, ($OUTPUT_FIELD_SEPARATOR), 729
$] ($PERL_VERSION), 728
$' ($POSTMATCH), 187, 228, 649, 730
$` ($PREMATCH), 187, 228, 649, 731
$$ ($PROCESS_ID), 90, 731
$(($REAL_GROUP_ID), 731
${^ (внутренние имена переменных), 90
$: ($FORMAT_LINE_BREAK_CHARACTERS), 282, 722
$0 ($PROGRAM_NAME), 731
$1 и т. п., 715
$^A ($ACCUMULATOR), 286, 716
вывод функции formline, 778
$a и $b (переменные сортировки), 716
$^C ($COMPILING), 718
$^D ($DEBUGGING), 544, 719
$^E ($EXTENDED_OS_ERROR), 721
$^F ($SYSTEM_FD_MAX), 474, 734
$^H (советы анализатору Perl), 723
$^I ($INPLACE_EDIT), 724
$^L ($FORMAT_FORMFEED), 284
$ ($LIST_SEPARATOR), 727
$^M (пул памяти), 727
$^O ($OSNAME), 678, 728
$ ($OUTPUT_RECORD_SEPARATOR), 548, 729
$^P ($PERLDB), 730
$^S ($EXCEPTIONS_BEING_CAUGHT), 720
$^T ($BASETIME), 137, 718
$^V ($PERL_VERSION), 730
$^W ($WARNING), 177, 735
${^WARNING_BITS}, 735
${^WIDE_SYSTEM_CALLS}, 454
$^X ($EXECUTABLE_NAME), 721
$[(индекс первой переменной в массиве), 716
в именах скалярных переменных, 36, 40, 87
в отладчике, 559
интерполяция скалярных переменных и, 97
метасимвол, 181, 200
при поиске, утверждение конца строки, 192, 222
разыменование с помощью, 299
символ прототипа, 272
+ (знак плюс), 134
++ (инкремента) оператор, 58, 128
+? квантификатор, 202
+= (присваивание сложения) оператор, 144
+ унарный оператор, 126, 130
аддитивный оператор, 132
в именах файлов, 810
квантификатор, 70, 201, 219
метасимвол, 181, 200
% (знак процента), 134
%= (присваивание взятия по модулю) оператор, 144
в именах хешей, 42, 88

в контрольных суммах, 885
взятия по модулю оператор, 131
обозначение типа переменной, 37
символ прототипа, 272
= (знак равенства)
= => маркер текущей строки, 559
= = (равно) оператор, 192,, 138, 640
=~ (привязка шаблона) оператор, 69, 130
использование с операторами поиска по шаблону, 184
= (присвоение) оператор, 37
=> (соответствует) оператор, 42, 114, 146
в директивах pod, 687
команда отладчика, 568
/ (косая черта), 134
//, 801
/= (присваивание деления) оператор, 144
деления оператор, 131
команда отладчика, 565
корневой каталог, 752
ограничители, замена, 98
' (одинарные кавычки), 96
в качестве ограничителей пакетов, 338
подавление интерполяции переменных и обработки escape-последовательностей трансляции, 37, 99, 236
рекомпиляция шаблона и, 189
сокращенные формы слов, как избежать смешивания, 248
" (двойные кавычки), 193, 183, 311
в аргументах formline, 778
в строке замены, 193
интерполяция переменных, 37
контекст, 108
обработка escape-последовательностей трансляции, 236
оператор преобразования в строку, 399
строки,использование в, 39
скалярные переменные, разыменование в, 194
_ (подчеркивание)
большие числа и, 95
в идентификаторах, 83, 90
в именах переменных, 339, 660
глобальный указатель файла, 715
изменения между версиями Perl, 645
использование в полях, 911
меченые данные и, 614
переносимость, 681
сдвоенное, 104
< (левая угловая скобка)
< (меньше) оператор, 53, 60, 138
<< во встроенных документах, 102,
для поиска по шаблону в <> 120,
<< (сдвиг влево) оператор 133, 1074
<<= (присваивание сдвига влево) оператор, 144
<=> (сравнение) оператор, 60, 138, 852
для выравнивания по левому краю, 285
в командах отладчика, 566,
в именах файлов, 810
оператор ввода строки, 53, 117
> (правая угловая скобка)
>= (больше или равно) оператор, 60, 138
> (больше) оператор, 53, 60, 138
>>= (присваивание сдвига вправо) оператор, 144
>> (сдвиг вправо) оператор, 133
в именах файлов, 810
в командах отладчика, 566
для выравнивания по правому краю, 285
для поиска по шаблону, 120
оператор ввода строки, 117
; (точка с запятой)
в именах файлов, угроза защите данных, 619
в командах отладчика, 560
в конце простых операторов, 150
ошибки, пропуск замыкающих, 640
простые команды, окончание, 83
встроенные документы, пропуск замыкающих, 640
^ (символ вставки)
^= (присваивание поразрядного XOR) оператор, 144
в тексте, вводимом в форматы, 281
инвертирование класса символов, 208
использование для, 217
как избежать особого использования в шаблоне, 208
метасимвол, 181, 200

объявление начала строки, 72, 192
поразрядного исключающего ИЛИ оператор, 139
при поиске по шаблону, 246
утверждение начала строки 73, 222
~ (тильда)
для подавления пустых строк, 282
поразрядного отрицания оператор, 129
расширение в исходный каталог, 788
. (точка), 134
.. (оператор диапазона), 141
... (оператор диапазона), 141
.= (конкатенация присваивания) оператор, 144
в @INC, 915
в классах символов, 217
в шаблонах, 245
векторы (IP-адреса), 781
групповой символ для символов, 206
в квантификаторах, 71
(конкатенация) оператор, 55, 132, 401, 652
метасимвол, 181, 200
поиск файлов по шаблону имени и, 788
разделение целых чисел в v-строках, 104
символ-заполнитель, 70
\ (обратная косая черта), 37
quotemeta функция для, 830
в escape-последовательностях, 96, 181
в многострочных командах отладчика, 558
ввод команд продолжения в отладчике, 560
для метасимволов, 201
для образования метапоследовательностей не-метасимволов, 181
для обхода метасимволов, 200
интерпретация анализатором регулярных выражений, 235
как разделитель каталогов в MS-DOS, 958
метасимволы, 181, 678
нотация для установки, 97
оператор ссылки, 130, 291, 304
строки, заключенные в кавычки, как избежать чрезмерного использования, 99
ссылки на найденный текст и, 74
: (двоеточие)
:: в идентификаторах пакетов, 101, 338, 645
:: в именах модулей, трансляция в разделители каталогов, 347
:: в полностью квалифицированных именах, 90
в XSUB, 586
в кластерах шаблонов, 230
в метках, 154
в объявлениях подпрограмм, 277, 278
в списках импорта модуля, 721
разрыв строки в форматах с $, 722
` (обратный штрих/обратные кавычки), 37, 98, 478
qx и, 830
защита данных и, 620
оператор, 108, 117, 644, 660
переносимость, 683
пример использования, 38
пустой контекст и, 660

Числа

-0 ключ командной строки, 539, 543
0+ (преобразование в число) оператор, 399
32- и 64-разрядные системы, 680

A

-a (autosplit) ключ командной строки, 544, 721
-A (возраст доступа) оператор проверки файла, 137, 718
abs (абсолютное значение) функция, 742
комплексные числа и, 969
перегрузка, 403
accept функция, 742
в preforked серверах, 472
использование с connect, 754
использование с модулем FileHandle, 873
пример использования, 492, 493, 966
сигналы и, 467
$ACCUMULATOR, 716
ActiveState дистрибутив Perl
PPM (Perl Package Manager), 604
модули только для Microsoft, 940

установка на системах Windows, 541
.al расширение имени файла, 345
alarm функция, 743
для установления тайм-аута поиска по шаблону, 638
чередование с sleep, 851
ALRM сигнал, 467
пример использования, 463
American Standard Code for Information Interchange, 1045
and (&&) логический оператор, 147, 661
приоритет, and и &&, 59, 362
AnyDBM_file модуль, 935
Apache веб-сервер, 591
CGI::Apache модуль, 935
mod_perl модуль расширения, 527
API с расширенными символами
разрешение Perl к использованию на целевой системе, 544
arctangent функция, 743
$ARG (английская $_), 117, 716
@ARGV массив, 717, 1050
eof и, 765
pop и, 826
-s ключ и, 551
shift и, 848
помеченные данные и, 612
пример использования, 65, 161, 198, 253, 486, 866
$ARGV переменная, 714, 1045, 1081
язык C в сравнении с Perl, 644
ARGV указатель файла, 714
ARGVOUT указатель файла, 718
ASCII, 82, 817, 1045
значения для символов, 817
преобразование в Unicode, 456
преобразование в символы, 752
async функция (в модуле Thread), 502
atan2 (арктангенс) функция
для вычисления π, 743
комплексные числа и, 969
$atime файловая статистика, 863
attributes прагма, 277, 901
attrs модуль (устарел), 936
AUTOLOAD подпрограмма, 166, 344, 376
использование с goto, 790
AutoLoader модуль, 345, 938
AutoSplit модуль, 345, 938
переносимость, 681
autouse прагма, 902, 938
AV (внутреннее значение массива), 583, 1045
awk, 68, 1045
преобразование в Perl, 115

B

b, 181, 203, 209, 223, 246
-b (блочный файл) оператор проверки файла, 135
-B (двоичный) оператор проверки файла, 137
B модуль, 939
B::Asmdata модуль, 939
B::Assemble модуль, 939
B::Bblock модуль, 939
B::Bytecode модуль, 528, 939
B::C и B::CC модули, 529, 939
B::Debug модуль, 939
B::Deparse модуль, 531, 940
B::Disassembler модуль, 940
B::Lint модуль, 530, 940
B::Showlex модуль, 940
b::Stash модуль, 940
B::Terse модуль, 940
B::Xref модуль, 531, 940
backspace, 96
base прагма, 371, 903, 936
реализация псевдохешей, 384
$BASETIME, 718
BASIC язык программирования, логические операторы из, 59
BEGIN блоки, 516
влияние на синтаксический анализ файла компилятором, 534
инициализация переменных перед вызовом подпрограмм, 269
настройка @inc, 347
порядок запуска, 533
Benchmark модуль, 939
сравнение времени выполнения альтернативных вариантов кода, 941
bind функция, 744
SO_REUSEADDR и, 848
пример использования, 492
binmode функция, 744, 815
sysopen и, 872
в связанных указателях файлов, 438
с seek и tell в DOS, 679
/bin/sh, 540
bless функция, 294, 305, 364, 746, 1066

typeglob и, 295
в сравнении с tie, 412
конструкторы, использование в, 365
неявное разыменование и, 290
пример использования, 662, 746, 833
blib каталог, 589
blib прагма, 904, 938
$blksize файловая статистика, 863
$blocks файловая статистика, 863
break команда (в C), 643
break оператор, 160
BSD (Berkeley Standard Distribution), 1045
BSD::Resource модуль, установка пределов используемых ресурсов для процессов, 638
ByteLoader модуль, 529, 940
bytes прагма, 455, 933

C

-C (время от последнего изменения дескриптора индекса) оператор проверки файла, 136, 137, 718
-c (проверка синтаксиса) ключ командной строки, 544, 718
Perl компилятор и, 520
-C (родные расширенные символы) ключ командной строки, 454, 544, 735
-c (символьный файл) оператор проверки файла, 135
/c модификатор шаблона, 198
продолжение поиска при отсутствии соответствия, 224
C язык, 1045
fcntl.h определения, загрузка как констант Perl, 934
Perl, использование из, 591
ввод и вывод в XSUB, 589
добавление интерпретатора Perl, 592
команды Perl, выполнение, 594
компиляция встроенных программ Perl, 592
подпрограммы Perl, вызов из, 593
стек Perl, обработка, 595
Perl, расширение с помощью, 582, 584, 586
внешние библиотеки C, использование функций из, 590
syslog функции, использование в Perl, 934
библиотека, сигналы, запускающие дампы памяти в, 464
библиотечные функции, 738
генераторы кода на C, 529
доступ к Perl из внешней среды, 527
заголовочные файлы, 1056
защита данных в сравнении с Perl, 611
операторы
логические, 140
отсутствующие в Perl, 148
связь с приоритетами, 125
определения типов, 583
поиск символа, 457
препроцессор, 550, 1070
программы оболочек, 623
программирование, отличия от Perl, 643
расширение Perl с помощью C, создание оболочек вокруг функций Perl, 585
статические переменные, 268
стек C, хранение переменных C, 526
структуры, 823
caller функция, 746
и стек контекста, 525
изменения между версиями Perl, 647
использование с goto, 166, 790
отладчик Perl и, 575
пример использования, 273, 389, 429, 700, 975
can метод (пакет UNIVERSAL), 374
Carp модуль, 416, 937, 943
case структура, 163
cat команда (Unix), 430
cbreak опция, 779
/cg модификатор шаблона, 192
CGI, 935
cgi.pm модуль, 944
mod_perl и, 26
безопасность сценариев, 494
защищенные разделы для, 630, 633
модули Perl для, 935
необходимость режима проверки меченых данных, 611
производительность под mod_perl, 527

сценарии, кажущиеся зависшими, 777
телеконференции по, 28
CGI::Apache модуль, 935
CGI::Carp модуль, 935, 944
CGI::Cookie модуль, 935
CGI::Fast модуль, 935
CGI::Pretty модуль, 935
CGI::Push модуль, 935
CGI-сценарии push-технологий, 935
charnames прагма, 456, 933
chdir функция, 748
в модуле Cwd, 947
выполнение при отсутствии аргумента, 555
пример замещения, 353
пример использования, 152, 620, 681, 761
приоритеты и, 126
CHECK блоки, 516
порядок запуска, 533
$CHILD_ERROR, 718
chmod функция, 51, 748
пример использования, 121, 737, 885
chomp функция, 54, 749
$/ и, 724
в сравнении с chop, 750
длина строки и, 458
$и, 548
пример использования, 644, 666, 755, 859
chop функция, 54, 750
длина строки и, 458
пример использования, 57, 942
эффективность, 650
chown функция, 751
в системах POSIX, 971
chr функция, 207, 752
CRLF и, 494
Unicode и, 214, 458
пример использования, 104, 403, 802
chroot функция, 630, 752
Class::Contract модуль, 390
Class::Multimethods модуль, 373
Class::Struct модуль, 931, 945
close функция, 479, 753
блокировка файлов и, 471
номера строк и, 724
отсоединение сокета с помощью, 754
переносимость, 682
пример использования, 53, 152, 334, 472
пример использования в каналах, 477, 478, 812
closedir функция, 754, 817
переносимость, 682
cmp оператор, 60, 138, 852
перегрузка, 403
cmp функция (в File::Compare), 954
comp.lang.perl телеконференции, 27
COMSPEC переменная оболочки, 556
Config модуль, 938
%config хеш, 678, 592
соответствие в операционной системе между именами и номерами сигналов, 464
connect функция, 754
использование IO::Socket вместо, 491
модуль Socket и, 973
пример использования, 491
constant прагма, 342
continue блок, 154, 159, 160
continue команда (в C), 643
Control-A, 194
Control-D как конец файла, 105, 674
Control-Z как конец файла, 105, 674
cookie (HTTP), установка и получение, 935
CORE псевдокласс, 372
CORE::GLOBAL псевдопакет, 353
cos (косинус) функция, 754
Math::Trig и, 969
комплексные числа и, 969
пример использования, 523, 744
CPAN (Comprehensive Perl Archive Network), 20, 46, 336, 346, 599, 1046
CPAN модуль, 938, 947
переносимость, 684
установка модулей с помощью, 604
modules каталог, 601
модули
декомпрессия и распаковка, 604
категории, 601
компиляция, 605
связывания для, 447
создание, 606
установка в библиотеку Perl, 605
установка и компиляция, 604

подкаталоги, 599
CPU
время использования процессами, 880
доступ в многозадачной среде, 622
освобождение протоками, 505
переменные условия, разрешающие потокам отдать его, 511
CRLF, 678
в программах для Интернета, 494
crypt функция, 755
пример использования, 112
$ctime файловая статистика, 863
<Ctrl>+<C>, 96
сигналы, обработчик для, 463
CV (внутреннее значение кода), 1046
Cwd модуль, 932
текущий рабочий каталог процесса, определение, 947

D

^D (Control-D) как EOF, 674
-d (каталог) оператор проверки файла, 61, 135
/d модификатор шаблона, 198
-d (отладка) ключ командной строки, 544, 557, 578
__DATA__ маркер (лексема), 105, 718
DATA указатель файла, 718
-D (установка флагов отладки) ключ командной строки, 544
Data::Dumper модуль, 334, 936, 948
date функция, 800
DB модуль, 936
caller функция, 747
DB_File модуль, 935
Data::Dumper модуль, использование с, 948
dbmclose функция, 412
dbmopen функция, 412
блокировка и, 473
СУБД и, 413
DBM-файлы, 1046
dbmclose и dbmopen функции, 756
блокировка, 473
модули для, 935
переносимость, 684
удаление из, 759
хранение сложных данных в, 447, 948
$DEBUGGING, 719
DEFAULT псевдосигнал, 464
defined функция, 757, 882, 1066
wantarray и, 267
в сравнении с exists и истиной, 770
массивы и, 108
оператор ввода строки и, 118
подпрограммы и, 264
пример использования, 340, 474, 476, 748, 756, 813
DEL в шестнадцатеричном виде, 96
delete функция, 759
в сравнении с undef, 883
пример использования, 302, 556
псевдохеши и, 303
связанные массивы, невозвращение удаленного значения для, 647
dequeue метод (Thread::Queue модуль), 513
detach метод, 504
$dev файловая статистика, 863
Devel::DProf модуль, 936
профилирование выполнения программ с помощью, 578
Devel::Peek модуль, 936
Devel::SelfStubber модуль, 938
Devel::SmallProf модуль, 581
diagnostics прагма, 936
__DIE__ ловушка, 733
die функция, 733, 760
alarm и, 468
END блоки и, 534
eval и, 766
sigtrap прагма и, 920
в обработчиках сигналов, 463
в сравнении с warn, 892
в сравнении с функцией exit, 772
обработчики сигналов и, 464
пример использования, 126, 140, 152, 657, 659
причина фатальных ошибок, 1081
установка значения выхода для, 728
dieLevel опция отладчика, 571
DNS (Domain Name Server), 489
do, 127
block, 762
FILE, 762
SUBROUTINE, 763
блок, 151
завершение выполнения, 162
итерация, 162
в сравнении с require, 762

команды управления циклами, 641
операторы управления циклом, 151
подпрограмма, 151
управление циклом, 162
doc каталог (CPAN), официальные страницы руководства Perl, 600
dosish.h файл, 540
dot-файлы, обработка с помощью связанных хешей, 428
down метод (Thread::Semaphore модуль), 514
dprof, 578
dprofpp программа, 578
dump функция, 763
Dumpvalue модуль, 936
dweomer, 1046
dwimmer, 1046
DynaLoader модуль, 345, 588, 938

E

-e (execute) ключ командной строки, 538, 546
-e (exists) оператор проверки файла, 61, 135
/e (вычисление выражений) модификатор шаблона, 194, 254
each функция, 329, 764
keys и, 794
пример использования, 878
связанные хеши и, 42, 432
эффективность, 654
else оператор, 63
elsif оператор, 63
emacs редактор
-*- последовательности, 539
поддержка отладчика, 569
регулярные выражения, использование в, 68
END блоки, 516
изменение кода завершения программы, 535
порядок запуска, 533
пропуск, 534
__END__ маркер (лексема), 104, 654, 674, 718
endgrent функция, 779
endhostent функция, 782
endnetent функция, 741
endprotoent функция, 785
endpwent функция, 785
endservent функция, 787
English модуль, 284, 936
enqueue метод (Thread::Queue модуль), 513
Env модуль, 936
env программа, запуск Perl через, 539
%ENV хеш, 719
переносимость, 682
удаление из, 759
EOF (конец файла), 1060
eof функция, 65, 653, 764
sysread и, 874
пример использования, 141, 160, 698
связанный указатель файла, использование с, 438
eq оператор, 60, 640
пример использования, 63, 67
equal метод (Thread модуль), 505
errno, 1063
Errno модуль, 936
escape-последовательности
для управляющих символов, 96, 236
трансляции в двойных кавычках, 236
/etc/group файл, 653, 779
/etc/hosts файл, 782
/etc/networks файл, 783
/etc/passwd файл, 653, 785
/etc/protocols файл, 785
/etc/services файл, 787
/etc/utmp файл, 782
eval exec, 51, 539, 551, 673
eval (Safe модуль), 632
eval метод в потоках, 503
eval функция, 92, 127, 288, 649, 766
AUTOLOAD, использование в, 344
caller и, 747
die функция и, 674, 760
do и, 762
qr и, 866
return и, 838
s///ee и, 195, 252
в циклах, 159, 162
во время компиляции, 516
возвращаемые ошибки, 720
возвращаемое значение, 668
восстановление структур данных с помощью, 334, 948
интерполированный код и, 636
исключительные ситуации и, 733
в потоках, использование с, 504

меченые данные, использование для, 613, 633
пакеты и, 339
перегрузка, изменение на этапе исполнения, 410
пример использования, 273
строки и, 172
трансляция с переменными, 199
форматы, использование в, 285
циклы и, 649
этап исполнения и, 518
$EVAL_ERROR, 720
eval_sv и eval_pv функции, 594
$EXCEPTIONS_BEING_CAUGHT, 720
exec функция, 768, 875, 1046
END блоки и, 518, 534
fork и, 770
замещение, возбуждение исключительных ситуаций, 952
переносимость, 683
пример использования, 613, 620
указатели файлов, сохранение открытыми между вызовами, 475, 734, 815
формат списка аргументов для избежания вызова оболочки, 635
$EXECUTABLE_NAME, 721
execv функция (в C), 769
execvp функция (в C), 768
exists функция, 664, 770
delete и, 759
вызов с хешем, 432
пример использования, 386, 389, 556
псевдохеши и, 303
связанные хеши и, 424
exit функция, 34, 540, 771, 776
END блоки и, 535
использование для пакетной обработки, 657
потоки и, 502, 504
пример использования, 474
exp функция, 772
пример использования, 48
expat анализатор XML, 669
Expect модуль, 483
@EXPORT массив, 721
Exporter модуль, 588, 938
закрытость модуля и, 349
export_fail метод, 352
@EXPORT_OK массив, 721
%EXPORT_TAGS хеш, 721
export_to_level метод, 351
$EXTENDED_OS_ERROR, 721
ExtUtils::Command модуль, 938
ExtUtils::Embed модуль, 592, 938
ExtUtils::Install модуль, 938
ExtUtils::Installed модуль, 938
ExtUtils::Liblist модуль, 938
ExtUtils::MakeMaker модуль, 607, 938
ExtUtils::Manifest модуль, 938
ExtUtils::Mkbootstrap модуль, 938
ExtUtils::Mksymlists модуль, 938
ExtUtils::MM_Cygwin модуль, 939
ExtUtils::MM_OS2 модуль, 939
ExtUtils::MM_Unix модуль, 939
ExtUtils::MM_VMS модуль, 939
ExtUtils::MM_Win32 модуль, 939
ExtUtils::Packlist модуль, 939
ExtUtils::Testlib модуль, 939

F

-f (время от последнего обращения к файлу) оператор проверки файла, 136
@F массив, 721
-f (обычный файл) оператор проверки файла, 61, 135
-F (шаблон для расщепления) ключ командной строки, 544, 546
fallback ключ перегрузки, 406
false значения, 38
FAQ, электронная документация Perl, 639
Fast CGI протокол, 935
Fatal модуль, 936
Fcntl модуль, 934, 953
fcntl функция, 772
Fcntl модуль и, 953
flock и, 775
h2ph и, 898
в сравнении с модулем FileHandle, 963
возвращаемое значение, 793
угроза защите данных, связываемая с, 629
флаг закрытия при exec, 629
использование, 475, 815
fields прагма, 371, 382, 936
объявления полей с помощью, 382
%FIELDS хеш, 722
FIFO, 484, 1046

__FILE__ лексема, 104, 674
File::Basename модуль, 681, 932
File::CheckTree модуль, 932
File::Chmod модуль, 749
File::Compare модуль, 932
File::Copy модуль, 932
File::DosGlob модуль, 932
File::Find модуль, 932
File::Glob модуль, 932
File::Path модуль, 932
File::Spec модуль, 681, 932
File::Spec::Functions модуль, 932
File::Spec::Mac модуль, 932
File::Spec::OS2 модуль, 932
File::Spec::Unix модуль, 932
File::Spec::VMS модуль, 932
File::Spec::Win32 модуль, 932
File::Stat модуль, 932, 960
  пример использования, 960
File::Temp модуль, ситуация гонки, возможность для, 628
fileglob оператор, 120
FileHandle модуль, 284, 287, 933
fileno функция, 438, 773
filetest прагма, 136
find функция, 69
  обход деревьев файлов, 932
FindBin модуль, 939
findstr функция, 68, 69
flock функция, 417, 470, 774
  alarms и, 468
for, 66, 150, 155
  как псевдоним foreach, 156
  поиск и замена элементов в массивах, 196
foreach циклы, 66, 150, 156, 649
  $_ и, 164
  изменения между версиями Perl, 647
  переменная цикла, программирование в Perl в сравнении с оболочкой, 644
  поиск и замена элементов в массивах, 196
fork функция, 479, 776, 1053
  $$ и, 731
  exec и, 770
  perlfork документация, 462
  shutdown и, 850
  socketpair и, 851
  wait и, 891
  блокировки, наследование в вызовах, 775
  в сравнении с моделью потоков, 499
  двунаправленная связь, 483
  дочерние процессы, наследование родительских указателей файлов, 473, 476
  использование для пакетной обработки, 657
  каналы и, 477, 826
  клонирование интерпретаторов в Windows, 498, 527
  открытие
    канала, предотвращение в, 621
    с ветвлением, 479, 480, 813
  переносимость, 683
  пример использования, 474
  процессы зомби и, 466
  серверы, клонирование себя с помощью, 493
format объявление, 777
$FORMAT_FORMFEED, 722
$FORMAT_LINE_BREAK_CHARACTERS, 722
$FORMAT_LINES_LEFT, 722
$FORMAT_LINES_PER_PAGE, 722
$FORMAT_NAME, 722
$FORMAT_PAGE_NUMBER, 722
$FORMAT_TOP_NAME, 723
formline функция, 286, 778
  $^a и, 716
FORTRAN, 83
freeware, 1052

G

-g (setgid) оператор проверки файла, 135
/g (глобальный) модификатор шаблона, 192, 194
GDBM (GNU DBM)
  GDBM_File модуль, 935
  блокировка файлов в, 473
ge оператор, 60
getc функция, 651, 779
  в связанных дескрипторах файлов, 436
getenv, 554
getgrent функция, 779
  замещение, 980
getgrgid функция, 780
getgrnam функция, 780

gethost* функции, замещение встроенных, 934
gethostbyaddr функция, 780
getpeername и, 783
замещение, 970
кэширование возвращаемых значений, 653
модуль Socket и, 973
обратное преобразование символов в байты, 453
пример использования, 493, 496, 787
gethostbyname функция, 781
замещение, 970
модуль Socket и, 973
пример использования, 493
gethostent функция, 782
переносимость, 782
getlogin функция, 782
getnet* функции, замещение встроенных, 935
getnetbyaddr функция, 783
getnetbyname функция, 783
getnetent функция, 783
Getopt::Long модуль, 932
Getopt::Std модуль, 932
getopts::* модули, 161
getpeername функция, 493, 783
пример использования, 974
getpgrp функция, 784
getppid функция, 784
пример использования, 796
getpriority функция, 784
setpriority, пример использования с, 847
getproto* функции, замещение встроенных, 935
getprotobyname функция, 784
модуль Socket и, 973
пример использования, 495
getprotobynumber функция, 785
getprotoent функция, 785
getpwent функция, 785
замещение, 981
пример использования, 112, 758
getpwnam функция, 786
замещение, 981
пример использования, 429, 751, 752
getpwuid функция, 782, 786
замещение, 981
использование для определения исходного каталога, 140
определение имени пользователя и информации, 906
пример использования, 429, 503, 755
getserv* функции, замещение встроенных, 935
getservbyname функция, 786
модуль Socket и, 973
пример использования, 495
getservbyport функция, 787
getservent функция, 787
getsockname функция, 787
getsockopt функция, 788
GID (ID группы), 480, 731, 751, 1046
поиск файлов по, 780
текущий, 719
$gid файловая статистика, 863
glob функция, 120, 121, 788
замещение, 354
пример использования, 788, 883
globbing filenames, 1046
gmtime функция, 789, 931
utime и, 888
замещение, 934, 979
и функция time, 879
инверсия, 979
пример использования, 824
goto функция, 165, 344, 649, 790
dump и, 763
изменения между версиями Perl, 646
использование с блоками, удаленными при оптимизации, 646
удаляемыми при оптимизации, 523
Greenwich Mean Time (GMT), 789
grep утилита, 69, 236, 1046
grep функция, 660, 791
$_ и, 664
каталог, использование с, 832
пример использования, 270, 354, 429, 442, 478, 493, 748
пустой контекст и, 660
собственного производства, 274
gt оператор, 60
GUI, использование Perl/Tk, 657
GV (внутреннее значение типа glob), 583, 1047

H

-h (help) ключ командной строки, 546

h2xs утилита, 585, 586, 607
here документы, 102
hex функция, 791
HOME переменная окружения, 555
hosts файл, 783
HTML (язык разметки гипертекста), 935
  pod-файлы, преобразование в, 937
  генерация с помощью модуля CGI, 935
  код, красиво форматированный, 935
HTTP (протокол передачи гипертекста)
  cookie, установка и получение, 935
  ссылки, поиск, 69
HV (внутреннее значение типа хеш), 583, 1047

I

-I (include path) ключ командной строки, 539, 548
-i (in-place editing) ключ командной строки, 546
/i (без учета регистра) модификатор шаблона, 188, 192, 194
I18N::Collate модуль, 933
ID пользователя, 751
ID процесса (PID)
  waitpid функция, 891
  возврат, 784
    с помощью fork, 776
  получение группы процесса для, 847
  получение группы процесса из, 784
if операторы, 60, 63, 150, 153
  C язык, отличия от Perl, 643
  в структуре case, 163
  логические операторы и, 58
IGNORE псевдосигнал, 464
import метод, 351, 792, 1057
  перегрузки константы, 407
@INC массив, 347, 723, 939
  прагма lib и, 914
  присоединение спереди каталогов, 548
%INC хеш, 347, 723
index функция, 792, 1058
  $, 716
  длина строки и, 458
  пример использования, 488
inet_ntoa функция (модуль Socket), 781
INIT блоки, 516
  инициализация переменных перед вызовом подпрограмм, 269
  порядок выполнения, 536
  порядок запуска, 533
init программа, 784
$ino файловая статистика, 863
$INPUT_LINE_NUMBER, 724
$INPUT_RECORD_SEPARATOR, 543, 724
int функция, 792, 822
  пример использования, 831
integer прагма, 131, 913
  пример использования, 900
I/O (Input/Output), 1047
IO (внутренний объект ввода-вывода), 1047
IO модуль, 933
IO::Dir модуль, 933
IO::File модуль, 933
IO::Handle модуль, 933
IO::Pipe модуль, 933
IO::Poll модуль, 933
IO::Pty модуль, 483
IO::Seekable модуль, 933
IO::Select модуль, 933
IO::Socket модуль, 490
IO::Socket модуль, 934, 966
IO::Socket::Inet модуль, 491
IO::Socket::INET модуль, 934
IO::Socket::UNIX модуль, 934
ioctl функция, 793
  h2ph и, 898
  возвращаемое значение, 793
  определение размера окна с помощью, 978
  пример использования, 793
IP (Internet Protocol), 489, 1047
  адреса
    использование нотации v-строк, 781
    представление в UTF-8, 453
    сетевые имена и, 783
IPC (межпроцессное взаимодействие), 461, 485
  в Интернете, 461
  каналы, 477
    двунаправленные, 482
  модули для, 933
  на одной машине, 462
  переносимость и, 683
  сигналы, 462
  сокеты в, 489
    передача сообщений, 495

сетевые клиенты, 491
сетевые серверы, 492
файлы как механизм, 469
IPC::Msg модуль, 934
IPC::Open2 модуль, 591, 934, 967
IPC::Open3 модуль, 591, 934, 967
IPC::Semaphore модуль, 934
IPC::Shareable модуль, 486
IPC::SysV модуль, 934
@ISA массив, 369, 726, 998, 1032, 1040
use base прагма и, 903
наследование через, 370
isa метод (пакет UNIVERSAL), 374
itimer программы, 743
IV (внутреннее целочисленное значение), 583, 1047

J

JAPH (Just Another Perl Hacker), 1047
Java, модули Perl для, 601
join метод (Thread модуль), 502
перехват исключительных ситуаций от, 503
join функция, 196, 794
в сравнении с split, 860
в сранении с pack, 824
пример использования, 101, 196, 305, 858
эффективность, 653

K

-k (липкий бит) оператор проверки файла, 135
keys функция, 42, 76, 115, 764, 794, 888
для подсчета записей в хеше, 797
пример использования, 270, 324, 888
связанные хеши и, 432
частые ошибки при использовании, 646
kill функция, 462, 795
для перечисления сигналов, 464
завершение обработки и, 535
пример использования, 465
системы Microsoft и, 465

L

-l (automatic line ending) ключ командной строки, 548
-l (символическая ссылка) оператор проверки файла, 135
last оператор, 67, 156, 159, 160, 164, 652, 660, 796
использование в do while, 162
пример использования, 120, 158, 665
управление циклом и, 162
$LAST_PAREN_MATCH, 727
lc функция, 797
пример использования, 76, 154, 165, 186
сортировка с помощью, 853
lcfirst функция, 797
le оператор, 60
length функция, 797
pos и, 827
байт-ориентированная оболочка для, 455
позиции символов и, 458
пример использования, 135
less прагма, 936
lib прагма, 347, 939
LIFO (Last In, First Out), 1047
#line директива, 674
_ _LINE_ _ маркер, 104, 674
LineInfo опция отладчика, 572
link функция, 798
lint (верификатор программ C), 530
list контекст
распознавание в отладчике, 559
listen функция, 798
пример использования, 492
$LIST_SEPARATOR, 727
local, 1062
дескрипторы файлов, 115
переменные, 169
присвоение обработчика сигнала, 464
local объявления, 169, 171, 798
в сравнении с my, 643
использование с глобальными переменными, 175
обычные ошибки программирования при использовании, 641
сохранение внутренних значений, 525
locale прагма, 933
localtime функция, 789, 800, 931
File::stat и, 865
strftime и, 970

замещение, 934, 979
инверсия, 979
пример использования, 111, 378, 945
функция time и, 879
lock функция, 506, 800
использование ссылки в качестве аргумента, 290
пример использования, 622
locked атрибут (подпрограммы), 277, 391, 509, 510
log (логарифм) функция, 801
комплексные числа и, 969
LOGDIR переменная окружения, 555
login имена функции для, 741
long, 822
long long, 822
lstat функция, 801
_ и, 137, 715
замещение, 932
lt оператор, 60
l-значения
методы, объявления, указывающие на возврат, 390
модификаторы и, 170
эффективность, 666

M

-M (use module) ключ командной строки
blib прагма и, 904
-m (use module) ключ командной строки, 549
-M (возраст модификации) проверка файла, 718
-m (время от последней модификации) оператор проверки файла, 136, 137
/m модификатор шаблона, 188, 192, 194
m// (поиск) оператор, 108, 180, 191, 565, 641, 801
=~ (привязка) оператор, использование с, 185
pos функция и, 827
интерполяция в двойных кавычках, обеспечение, 183
модификаторы, 188, 191
m?? (поиск по шаблону) оператор, 193, 565
reset функция и, 838
Macintosh операционная система
File::Spec:, 932
интерфейс командной строки, 50
использование кавычек, 542
вызов приложений Perl, 540
magic, 1054
mail, отправка, переносимость, 683
main пакет, 92, 337
Makefile, 1047
создание для расширений Perl, 938
Makefile.PL, 586, 604, 606
MakeMaker, 607
malloc
PERL_DEBUG_MSTATS переменная окружения, использование с, 556
ошибки, 727
man команда, 21, 346, 568, 1047
MANIFEST файл, 587, 938
map функция, 76, 660, 791, 801
перевод в верхний регистр с помощью, 435
преобразование Шварца и, 855
пример использования, 76, 238, 334
пустой контекст и, 660
$MATCH, 728
Math::BigFloat модуль, 931
Math::BigInt модуль, 395, 931
Math::Complex модуль, 931
Math::Trig модуль, 931
method атрибут (подпрограммы), 277, 391
Method (в прагме overload), 409
m//g, изменения между версиями Perl, 647
Microsoft Windows
File::Spec:, 932
Win32::Pipe модуль, 484
ветвление и клонирование интерпретаторов в, 498
дистрибутив Perl ActiveState, установка, 541
использование кавычек в NT, 542
модули CPAN, установка в, 604
модули Perl для, 940
оболочка, установка альтернативной для использоввания Perl, 556
перенос информации (perlwin32), 462
поиск файлов с помощью File::DosGlob, 932
реестр, обработка, 448
сигнал номер 0 в, 465
системные вызовы, использующие API расширенных символов, 454

mkdir функция, 802
в сравнении с системным вызовом mkdir, 653
пример использования, 661
mkfifo функция, 484
MLDBM модуль, 447
Data::Dumper модуль, использование с, 948
mmap системный вызов, 485
$mode файловая статистика, 863
mod_perl, 527, 591
modules каталог модулей (CPAN), 600
MS-DOS операционная система, вызов программ Perl, 540
msgctl функция, 803
msgget функция, 803
msgrcv функция, 803
msgsnd функция, 803
$mtime файловая статистика, 863
my функция, 91, 149, 157, 169, 171, 804
в сравнении с local, 643
объявления, 804
замыкания и, 306
пакеты и, 337
переменные управления циклом и, 154
правила контекста и, 106
пример использования, 268
присвоение списков, 112
сравнение с local, 176
сравнение с our, 91
частые промахи при использовании, 641
эффективность, 652

N

-n (неявный цикл) ключ командной строки, 550
names
сетевые адреса, получение из, 780
сигнал, 464
NDBM_File модуль, 935
ne оператор, 60, 640
Net::hostent модуль, 782, 934
Class:, 946
Net::netent модуль, 783, 935
Net::Ping модуль, 935
Net::proto модуль, 785
Net::protoent модуль, 935
Net::servent модуль, 935
networks файл, 783
new метод, 502, 807
next оператор, 67, 154, 159, 160, 807
next if конструкция, 650
в do { } while, 162, 643
для выхода из однократно выполняемых блоков, 162
NFS (сетевые файловые системы), 1047
существующие ограничения, 489
$nlink файловая статистика, 863
nn метазнак, 206
no объявления, 166, 176, 348, 420, 808
nomethod ключ перегрузки, 406
null
дескриптор файла, 119
значения, 38
null device (фиктивное устройство), Unix, 444
NV (внутреннее значение типа действительное число), 583

O

O модуль, 528, 530, 940
/o (однократно) модификатор шаблона, 188, 189, 192, 194, 234
-O (реальный UID) оператор проверки файла, 135
-o (текущий UID) оператор проверки файла, 135
oct функция, 95, 748, 808
chmod, использование с, 748
пример использования, 95
Opcode модуль, 936
open прагма, 917
open функция, 52, 479, 808, 871
$. и, 753
$^F и, 734
fileno функция и, 773
open прагма и, 917
open2 и open3 в сравнении с, 482
set-id программы и, 619
блокировка файлов и, 471
в сравнении с binmode, 744
в сравнении с sysopen, 871
в сравнении с модулем FileHandle, 962
временные файлы и, 626
для канала, 477
формат с двумя аргументами, 812
защита данных и, 612
именованные каналы и, 485

конвейеризованная
риск для защиты данных, 618
формат списка аргументов для избежания вызова оболочки, 635
открытие с ветвлением, 479, 813
передача указателей файлов с помощью, 474
переносимость, 681, 683
преобразование дескриптора файла в указатель, 476
пример использования, 120
режимы в формате трех аргументов, 810
связанные дескрипторы файлов и, 434
сетевые серверы и, 493
список файлов и, 295
Open2, Open3 библиотечные модули, 482
opendir функция, 817, 832
ops прагма, 936
or оператор, 139, 140, 141, 147, 661
| (поразрядный), 139
использование со списковыми операторами, 167
приоритет, or и ||, 59, 362
ord функция, 752, 817
Unicode и, 458
переносимость, 685
OS/2 операционная система
File::Spec:, 932
вызов приложений Perl, 540
$OS_ERROR, 728
$OSNAME ($^O переменная), 728
our объявление, 91, 149, 169, 171
в сравнении с my, 804
глобальные переменные с лексической областью видимости, 173
основания для ввода, 178
правила контекста и, 106
прагмы subs и vars в сравнении с, 901
прагма strict и, 924
прагма vars и, 925
$OUTPUT_AUTOFLUSH, 729
$OUTPUT_FIELD_SEPARATOR, 729
$OUTPUT_RECORD_SEPARATOR, 729
overload прагма, 396

P

-p (вывод во время цикла) ключ командной строки, 550
-p (именованный канал) оператор проверки файла, 135, 484
-P (препроцессор C) ключ командной строки, 550
pack функция, 680, 819, 884
UTF-8 и, 458
vec и, 890
длина строки и, 458
порядок байтов и, 680
пример использования, 104, 804, 846, 870
символы шаблона для, 820
package объявление, 824
pads, 88
PalmPilot, модули Perl для, 601
paragraph mode, 543
PATH переменная окружения, 51, 551, 555
защита данных и, 618
PAUSE, 606
Perl
C, использование из, 584
расширение с помощью XSUB, 589
библиотека, установка модулей CPAN в, 605
версии, 728, 730
виртуальная машина, 524, 526
встраивание в C, 591
интерпретатор Perl, добавление в программу C, 592
команды Perl, выполнение из C, 594
компиляция встроенных программ, 592
подпрограммы Perl, вызов из C, 593
стек Perl, обработка из C, 595
использование Unicode, предостережения по, 459
использование из C
XSUB и язык XS, 585
функций из внешних библиотек C, 590
создание расширений, 586
использование специальных символов, 36
история, 703
местонахождение в вашей системе, 542
программы, жизненный цикл, 516

программирование
потоков, 501
часто используемые приемы, 639
расширения и свойства, модули для, 936
черный, 706
Perl Data Language (PDL) модуль, 319
Perl Package Manager (PPM), 604
Perl интерпретатор, 50, 516
-s ключ, 557
липкий (sticky) бит, 654
PERL5DB переменная окружения, 555
отладчик, настройка с помощью, 570
PERL5LIB переменная окружения, 555
PERL5OPT переменная окружения, 555
PERL5SHELL переменная окружения, 556
perlapi документация, 594
perlcall документация, 595
perlcc команда, 528, 718
получение pbyscript, 529
$PERLDB, 730
PERLDB_OPTS переменная окружения
AutoTrace опция, трассировка команд фазы компиляции, 560
использование при настройке отладчика, 569
PERL_DEBUG_MSTATS переменная окружения, 556
PERL_DESTRUCT_LEVEL переменная окружения, 379, 556
perldoc команда, 21, 346
вызов отладчиком, 568
perlembed документация, 591
eval_sv функция, 595
perlfork документация, 462
perllocale документация, 797
perlmodinstall документация, 604
perlport документация, 462
информация о переносимости функций, 742
Perl/Tk, 657
$PERL_VERSION, 728, 730
perlwin32 документация, 462
PID (ID процесса), 731, 784, 815
получение группы процесса из, 784
pipe функция, 826
socketpair в сравнении с, 483
Win32 и, 484
в сравнении с socketpair, 851
переносимость, 683
.pm файлы, 336
загрузка связанных файлов с помощью, 348
pod, 686
встраивание документации в программу Perl, 84
директивы, 689, 1056
модули, 695
модули Perl для, 601
трансляторы, 687, 695
Pod::Checker модуль, 937
Pod::Functions модуль, 937
Pod::Html модуль, 937
Pod::InputObjects модуль, 937
Pod::Man модуль, 937
Pod::Parser модуль, 937
Pod::Select модуль, 937
Pod::Text модуль, 937
Pod::Text::Termcap модуль, 938
Pod::Usage модуль, 938
pop функция, 315, 826, 849, 933
splice и, 856
split и, 857
пример использования, 111, 270, 669
связанные массивы и, 426
popen функция (язык C), 591
pos функция, 224, 827
@+ и, 726
длина строки и, 458
POSIX модуль, 934
getattr функция, 779
strftime функция, 800
блокировка сигналов с помощью, 468
получение экспортируемых функций и переменных с помощью, 348
POSIX-стиль классов символов, 208, 217
$POSTMATCH, 730
PPM (Perl Package Manager), 604
$PREMATCH, 731
print функция, 53, 76, 652, 828, 829
select и, 843
в сравнении с read, 831
в сравнении с write, 283, 894
в сценариях CGI, 673
завершение записей с помощью $\, 729

замещение, возбуждение исключительных ситуаций, 952
использование скобок, 134
объекты и, 400
обозначение косвенных объектов, 361
очистка буфера с промощью $|, 729
ошибка запятой, 640
разделение полей с помощью $?@, 729
связанные дескрипторы файлов и, 435
эффективность, 649, 652
printf функция, 143, 829, 860
$# и, 716
замещение, возбуждение исключительных ситуаций, 952
пример использования, 754, 781, 864, 880
связанные указатели и, 437
стиль и, 672
эффективность, 649
PrintRet опция отладчика, 572
$PROCESS_ID, 731
$PROGRAM_NAME, 731
prototype функция, возбуждение исключительной ситуации, 952
push функция, 86, 315, 830, 849
pop функция, 87
splice и, 856
пример использования, 270, 317
связанные массивы и, 425
ссылки на массивы и, 298
хеши и, 41
эмуляция, 271
push-pop (pp) коды, 524
PV (внутреннее значение типа строка), 583
pwd команда (Unix), 654
Python, 83
отличия от Perl, 640

Q

q// (кавычки)
оператор, 108, 185, 237, 408
модификаторы для, 188
quotemeta функция, 830
пример использования, 259
qw (заключение в кавычки слов) оператор, 100, 111
qx// (выполнение в кавычках) оператор, 108, 117
open прагма и, 917
readpipe и, 834

R

-R (доступен по чтению с реальными uid/gid) оператор проверки файла, 135
-r (доступен по чтению с текущими uid/gid) оператор проверки файла, 61, 135
rand функция, 126, 831, 863
пример использования, 126, 166, 403, 636, 755
$rdev файловая статистика, 863
re прагма, 936
read функция, 831, 832
в сравнении с print, 831
связанные указатели и, 437
сигналы и, 467
readdir функция, 832
ReadLine модуль, 574
readline функция, 117, 833
readlink функция, 833
readpipe функция, 834
recallCommand опция отладчика, 571
record mode, 725
records
output separator, 729
recv функция, 834, 846
пример использования, 496
сокеты и, 846
redo оператор, 159, 160, 834
ref функция, 305, 365, 835
attributes прагма и, 902
пример использования, 366, 386, 397, 401, 417
reftype функция (attributes прагма), 365
regex, 68
regexp, 68
rename функция, 836
переносимость, 682
пример использования, 547, 767
require функция, 168, 348, 723, 836, 886
base прагма и, 903
caller и, 747
__data__ и, 105
@INC и, 897
package объявления и, 825
tie и, 413

библиотеки Perl и, 896
в сравнении с do, 762
в сравнении с use, 886
загрузка функций и, 269
объявления пакетов и, 338
области видимости и, 172
прагма lib и, 914
res, 68
reset функция, 838
return функция, 159, 265, 838
eval и, 162, 766
завершение потоков, 502
ошибки и, 267
reval метод, выполнение подозрительного кода с помощью, 631
reverse функция, 76, 839
scalar и, 459
запрещение с sort, 647
пример использования, 76, 156, 402
rewinddir функция, 817, 839
rindex функция, 839
длина строки и, 458
rmdir функция, 840
root пользователь, 1078
rot13 шифрование, 198
RS (переменная awk), 724
rstart (переменная awk), 726
RV (внутреннее значение типа ссылка), 583

S

-s (switch) ключ командной строки, 551, 557
/s модификатор шаблона, 188, 192, 194, 198
-s (ненулевой размер) оператор проверки файла, 135
s/// (подстановка) оператор, 108, 180, 193, 641, 1069
=~ и !~ операторы привязки, 185
интерполяция в двойных кавычках, обеспечение, 183
модификатор шаблона для, 188, 194
-S (поиск сценария) ключ командной строки, 551
-S (сокет) оператор проверки файла, 135
Safe модуль, 630
работа с ненадежным кодом, 630
safe модуль, 936
scalar контекст
распознавание в отладчике, 559
scalar функция, 106, 840
reverse и, 459
массивы и хеши, определение размера, 797
пример использования, 305
прототипы и, 271
scripts
каталог CPAN, 600
SDBM_File модуль, 935
Search::Dict модуль, 931
sed, 68, 197
seek функция, 841, 877
Fcntl модуль и, 953
File::Temp и, 629
-x ключ и, 554
пример использования, 472
связанные указатели и, 437
текстовый режим DOS и, 679
seekdir функция, 817, 842, 877
telldir и, 877
select (готовые дескрипторы файлов) функция, 773, 843, 844
fileno и, 774
IPC::Open3 и, 968
быстрое засыпание с помощью, 845, 851
select системный вызов, мультиплексирование ввода/вывода между клиентами, 493
select (указатель выходного файла) функция, 828, 843, 933
IPC::Open3 и, 482
FileHandle модуль как замена, 962
быстрое засыпание с помощью, 485
для переменных формата, 284
переменные и, 712
пример использования, 284, 495, 548, 814, 893
SelectSaver модуль, 933
self метод (Thread модуль), 504
SelfLoader модуль, 345, 939
Devel::SelfStubber модуль, использование с, 938
semctl функция, 845
semget функция, 846
semop функция, 846
send функция, 846
setgid, 480, 610, 619
setgrent функция, 779
sethostent функция, 782

set-id программы, 610
опасные операции, выполнение, 619
ошибки в ядре Unix, проблемы защиты данных, 623
setnetent функция, 741, 783
setpgrp функция, 465, 847
setpriority функция, 847
setprotoent функция, 785
setpwent функция, 785
setservent функция, 787
setsockopt функция, 848
пример использования, 492, 495
setuid, 480, 610, 619
sh (оболочка Борна), 554, 1064
#! нотация и, 540
eval exec прием, 51, 551
exec и, 768
system функция и, 612
User::pwent и, 981
-x эмуляция в Perl, 545
каналы и, 812
коды завершения, 875
конвейеры и, 477
shebang, нотация, 50
Shell модуль, 934
shift функция, 65, 315, 848, 849, 885
@_ и, 664
splice и, 856
очереди и, 830
пример использования, 805
эффективность, 649
shmctl функция, 849
пример использования, 488
ShMem пакет, 487
shmget функция, 849
пример использования, 487
shmread функция, 849
пример использования, 488
shmwrite функция, 849
пример использования, 488
short, 861
shutdown функция, 850
отсоединение сокета с помощью, 754
пример использования, 851
%SIG хеш, 462, 732
sigaction системный вызов, 467
SIGALRM сигнал, 743
signalLevel опция отладчика, 571
sigprocmask системный вызов, 468
sigtrap прагма, 464
sin (синус) функция, 850
комплексные числа и, 969
пример использования, 523, 744
$size файловая статистика, 863
sleep функция, 851
alarm и, 743
select функция вместо, 845
пример использования, 156
разрешение точнее секунды, 845
slurping файл, 543
Socket модуль, 490, 935
inet_ntoa функция, 781
сетевой клиент, соединение с сервером, 491
socket функция
accept и, 742
самооживление с помощью, 799
socketpair функция, 483, 851
sort функция, 42, 76, 852
reverse и, 647
длина строки и, 458
локализация $a и $b, 712
национальные установки и, 916
оптимизация, 522
переменные, 716
сравнения операторы в, 60
хеши и, 327
spaceship (космический корабль) оператор (), 139
splice функция, 856
в сравнении с delete, 759
изменения между версиями Perl, 646
пример использования, 403, 420, 827, 830
связанные массивы и, 426
эффективность, 649
split функция, 68, 185, 196, 229, 857
join и, 196
reverse и, 402
замещение, возбуждение исключительных ситуаций, 952
поля переменной ширины, извлечение с помощью, 1069
пример использования, 49, 112, 151, 323, 325, 663, 802
шаблоны и, 68
эффективность, 650, 653
sprintf функция, 437, 791, 860
двухзначные годы и, 789
длина строки и, 458
нотация вектора с точками и, 781
округление с помощью, 792

пример использования, 195, 414, 808
связанные дескриптоы и, 433
связанные указатели и, 437
sqrt (квадратный корень) функция, 862
комплексные числа и, 969
пример использования, 850
srand функция, 863
src каталог (CPAN), 600
stat функция, 801, 863, 960
_ и, 137, 715
File::Stat и, 616
в сценариях суперпользователя, 136
вызов с указателями файлов, 626
замещение, 932
права доступа к файлам и, 808
пример использования, 111
STDERR указатель файла, 52, 662, 733, 774
отправка выдачи в другой процесс, 448
передача указателя файла новым программам через, 474
переназначение, 494
STDIN указатель файла, 52, 733, 774
-i ключ командной строки и, 548
передача указателя файла новым программам через, 474
переназначение, 494
чтение одного символа из, 779
STDOUT указатель файла, 52, 53, 733, 774, 1078
-i ключ командной строки и, 548
вывод по, 828
передача указателя файла новым программам через, 474
переназначение, 494
strict прагма, 47, 91, 178, 310, 922, 936
голые слова и, 642
struct эмуляция, 384
Struct::Class модуль, 381
StrVal (в прагме overload), 409
study функция, 865
sub объявление, 262, 272, 294, 867
subs прагма, 377, 936
замещение встроенных функций с помощью, 353
$SUBSCRIPT_SEPARATOR (*см. также* S;), 115, 734
substr функция, 868
chop функция и, 750
длина строки и, 458
манипуляции позициями в строке, 222
при укорачивании строк, 549
пример использования, 401, 488, 699, 750
специальные переменные в сравнении с, 726
эффективность, 651, 654
SUPER псевдокласс, 372
SV (внутреннее значение типа скаляр), 583
SWIG, система, автоматическая генерация XSUB, 585
switch оператор, 163, 652, 1059
Perl в сравнении с программированием C, 643
symlink функция, 869
переносимость, 681
Sys::Hostname модуль, 934
Sys::Syslog модуль, 934
syscall функция, 743, 869
fileno и, 774
h2ph и, 898
setitimer и, 851
возвращаемое значение, 738
syslog, привязка дескриптора файла к, 448
sysopen функция, 472, 815, 871
Fcntl модуль и, 953
open прагма и, 815, 917
блокировка файлов и, 471
меченые данные и, 619
перезапись файлов и, 625
предотвращение ситуации гонок с помощью, 485
пример использования, 619, 627, 970
разрешения доступа к файлам и, 882
sysread функция, 832, 874
seek и, 841
в сравнении с read, 832
пример использования, 876, 972
связанные указатели и, 437
System V IPC, 485
IPC::SysV модуль, 934
переносимость, 683
системные вызовы, 462
ненадежность сигналов, 467
совместно используемая память, функции для, 849

сообщения, системные вызовы для, 803
убиение групп процессов, 796
функции для, 740
функции для семафоров, 845
system функция, 768, 875
$? и, 535, 718, 760
exit и, 776
set-id программы и, 619
wait и, 891
в сравнении с exec, 768
в сравнении с обратными штрихами, 660
замещение, возбуждение исключительных ситуаций, 952
меченость и, 613
оболочки Windows и, 556
переносимость, 683
сравнение с syscalls, 24
формат списка аргументов для избежания вызова оболочки, 635
эффективность, 654
$SYSTEM_FD_MAX, 734
syswrite функция, 433, 876
seek и, 841
обработчики сигналов и, 732
пример использования, 418
связанные указатели и, 433, 438

T

-t (tty) оператор проверки файла, 135
-T (проверка меченых данных) ключ командной строки, 490, 610, 552
серверы и, 494
-T (текст) оператор проверки файла, 61, 137
taintperl программа, 647
tan (тангенс) функция, 743, 969
Math::Trig и, 969
TCP, 495, 934, 1049
IO::Socket:, 491, 492
Socket модуль и, 491
в сравнении с UDP, 495
клиенты/серверы, 491
сокеты и, 489
через сокеты домена Интернета, 489
tee программа, Unix, 444
tell функция, 877
fgetpos как альтернатива, 963
пример использования, 472, 842
связанные указатели и, 437
текстовый режим DOS и, 679
telldir функция, 817, 877
seekdir и, 877
Tengwar письмо, 457
Term::Cap модуль, 935
Term::Complete модуль, 935
Term::ReadKey модуль, 569, 779
Term::ReadLine модуль, 569, 935
отключение поддержки отладчика, 574
Term::Rendezvous модуль, 574
test команда (Unix), 51
Test модуль, 939
Test::Harness модуль, 939
test.pl, 586, 590
Text::Abbrev модуль, 931
Text::ParseWords модуль, 931
Text::Wrap модуль, 931
Thread модуль, 501
detach метод, закрытие потоков с помощью, 504
join метод, перехват исключительных ситуаций от, 503
идентификация потоков, 504
передача процессорного времени, 505
создание потоков, 502
уничтожение потоков, 502
Thread::Queue модуль, 512
Thread::Semaphore модуль, 501, 511, 513
Thread::Signal модуль, 514
tid метод (Thread модуль), 504
tie функция, 411, 877, 879
DBM-файлы и, 473
use или require и, 413
в сравнении с bless, 412
ловушка при отвязывании, 444
модули CPAN для, 447
переносимость, 682
пример использования, 313, 948
Tie::Array модуль, 421, 937
splice подпрограмма, использование, 426
Tie::Counter модуль, 419
Tie::DBI модуль, 447
Tie::DevNull модуль, 443
Tie::DevRandom модуль, 443
Tie::Hash модуль, 427, 937
Tie::Persistent модуль, 447
Tie::Refhash модуль, 937

Tie::Scalar модуль, 414, 937
Tie::SecureHash модуль, 390, 448
Tie::StdArray модуль, 421
Tie::STDERR модуль, 448
Tie::StdHash модуль, 427
Tie::StdScalar модуль, 414
Tie::SubstrHash модуль, 937
Tie::Syslog модуль, 448
Tie::Tee модуль, 443
Tie::TextDir модуль, 448
Tie::TransactHash модуль, 448
Tie::VecArray модуль, 448
Tie::Watch модуль, 448
tied функция, 879
    пример использования, 431
time функция, 789, 879, 931
    переносимость, 684
    преобразование в поля, 789, 800
Time::gmtime модуль, 934
Time::Hires модуль, 743
Time::localtime модуль, 934
Time::tm модуль, 934
timelocal подпрограмма, 800
times функция, 880
Tk, 657
tk опция отладчика, 571
tr/// (транслитерация) оператор, 184, 197, 880, 894
    =~ и !~ операторы привязки, 185
    модификаторы для, 188
    символы, транслитерация с помощью, 458
troff, 1049
true значения, 38
truncate функция, 880
    пример использования, 472
try блок, 766
tty, 483
typedef (C язык), 1049
typeglob, 36, 89, 115, 288
    local функция, использование с, 799
    анонимные, 295
        генерирование в модуле Symbol, 931
    в сравнении с ссылками на typeglob, 295
    в хешах таблиц имен, 340
    вывод содержимого, 573
    переменные, связывание с классами через, 412
    подпрограммы, разрешение в, 359
    самооживление, 435
    символ прототипа (*), 37
    создание псевдонимов в таблице имен, 342
    ссылки на дескрипторы файлов, создание, 295
typeglobs, 305
    IO::Handle объект, 296
typemap файл, 586

U

-u (setuid) оператор проверки файла, 135
-U (unsafe) ключ командной строки, 552
-u (дамп памяти) ключ командной строки, 552
uc функция, 458, 881
    пример использования, 389, 435, 954
    сортировка с помощью, 853
ucfirst функция, 458, 881
    пример использования, 380, 981
UDP, 1049
    в сравнении с TCP, 495
    пакеты, 489
    передача сообщений с помощью, 495
    пример использования, 495
    сокеты и, 489
UID (ID пользователя), 480, 751
    запись в файле passwd, 786
    текущий и реальный, 719
$uid файловая статистика, 863
umask функция, 881, 1049
    mkdir и, 802
    open и, 811
    sysopen и, 872
    пример использования, 620
undef (значение), 62, 107, 111, 757
    select и, 485, 845
undef функция, 62, 882
    массивы, освобождение памяти, 113
    пример использования, 760, 771
    проглатывание файлов с $/ и, 334, 725
    хеши и, 795
    эффективность, 653, 654
Underscore модуль, 420
Unicode, 82, 451
    база данных свойств, 459
    верхний регистр, преобразование строк в, 199

в идентификаторах, 90
групповые метазнаки, поиск с помощью, 207
использование в Perl, предостережения по, 459
национальные установки и, 797
поддержка, включение, 933
преобразование в символы, 752
редакторы для, 456
сведения о, 211
свойства, 208, 210
блоков, 214
двунаправленный вывод, 213
декомпозиция символов, 213
слоговые азбуки, классификация по гласным звукам, 214
сокращенные обозначения классов символов в Perl и, 209
составные в Perl, 211
стандартные, 212
сводка, доступ, 459
символ-разделитель абзацев, 83
символ-разделитель строк, 83
(слово) метасимвол, 70
смайлик, 96, 752
Стандарт Unicode, версия 3.0, 459
таблицы перевода регистра, 458
UNIVERSAL класс, 374, 937
методы, добавление в, 376
Unix, 37, 1049
#! и, 50
File::Spec:, 932
fork на, 777
IPC и, 462
test команда, контрольные сценарии и, 51
вызов Perl и, 537
время в, 879
дескрипторы файлов, 774
замещение функций для эмуляции, 353
использование кавычек, 541
командная строка, 29
конец файла в, 674
модули CPAN, установка в, 604
оболочки, 37
ошибки защиты, 623
перевод строки в, 678, 815
поиск файлов по шаблону имени, 932
права доступа, 882
происхождение Perl и, 13
пути к файлам в, 681
пути поиска, 552
сигналы и, 465, 467
символы перевода строки в, 745
сокеты и, 489, 934
усечение интерпретации ядром строки #!, 539
установка Perl, 20
фигурные скобки в, 311
функции в Perl, 738, 742
эмуляция в Windows, 940
unless оператор, 64, 150
в сравнении с операторами if, 153
unlink функция, 883
модуль Socket и, 974
переносимость, 682
пример использования, 107, 431
unpack функция, 680, 820, 884
UTF-8 и, 458
vec и, 890
длина строки и, 458
использование с полями фиксированной ширины, 1069
переносимость, 656
порядок байтов и, 680
пример использования, 756, 793, 817, 863, 870, 885
эффективность, 651
unshift функция, 315, 849, 885
splice и, 856
пример использования, 699, 737, 799
unsigned short, 861
untie функция, 411, 446, 756, 886
ожидающие обработки ссылки, сохранение, 414
пример использования, 473, 878
until циклы, 150, 154
up метод (Thread::Semaphore модуль), 514
use, 46, 166, 168, 176, 336, 348, 886
tie функция и, 413
глобальная и лексическая область видимости, 169
usenet
Perl, телеконференции, 27
User::grent модуль, 780, 934
User::pwent модуль, 786, 934
Class:, 946
UTF-8, 82, 207, 210, 453

binmode и, 744
дисциплины ввода/вывода и, 816, 917
переносимость, 656, 680
преобразование в, 669
преобразование в байты/из байтов, 454
преобразование в целые числа, 458
смайлик и, 104
сценарии CGI и, 633
эффективность, 651
utf8 прагма, 207, 210, 455, 933
глобальное переключение интерфейсов на, 735
национальные установки и, 218
пример использования, 216
utime функция, 879
utmp файл, 783
UV (внутреннее значение типа целое без знака), 583

V

-V (версия) ключ командной строки, 553
-v (версия) ключ командной строки, 553
values функция, 764, 888
связанные хеши и, 427, 432
vars модуль(устарел), 936
vec функция, 889
длина строки и, 458
пример использования, 496, 844
эффективность, 654
VERSION метод, 375
vi редактор
поддержка отладчика, 569
регулярные выражения, использование в, 68
VMS операционная система
File::Spec:, 932
вызов приложений Perl, 540
void контекст
распознавание в отладчике, 559
v-строки (вектор или версия), 104, 1049
in $^V, 730

W

-W (доступен по записи с реальными uid/gid) оператор проверки файла, 135
-w (доступен по записи с текущими uid/gid) оператор проверки файла, 61, 135
-w ключ командной строки, 171, 177, 553, 721, 982
включение вывода предупредительных сообщений, 51
wait функция, 890
$? и, 718
die и, 753
fork и, 777
процессы зомби и, 466
waitpid функция, 890, 891
close и, 753
IPC::Open2 и, 482, 967
IPC::Open3 и, 967
пример использования, 474, 483, 826, 971
процессы-зомби и, 466, 890
wantarray функция, 106, 267, 891, 266
пример использования, 481, 883
__WARN__ ловушка, 732
прагма diagnostics и, 909
пример использования, 892
warn функция, 892, 943
номера строк и, 674
пример использования, 161
сценарии CGI и, 944
$WARNING, 735
${^WARNING_BITS}, 735
warnings прагма, 177, 557, 658, 926, 937
__WARN__ и, 733
warnLevel опция отладчика, 571
WeakRef модуль, 379
while цикл, 60, 64, 150, 154
eof функция в, 764
булев контекст, создание, 107
оператор угловых скобок и $_, 117
присвоение списку в, 112
${^WIDE_SYSTEM_CALLS}, 735
Win32::ChangeNotify модуль, 940
Win32::Console модуль, 940
Win32::Event модуль, 940
Win32::EventLog модуль, 940
Win32::File модуль, 940
Win32::FileSecurity модуль, 940
Win32::Internet модуль, 940
Win32::IPC модуль, 940
Win32::Mutex модуль, 940
Win32::NetAdmin модуль, 940
Win32::NetResource модуль, 940
Win32::ODBC модуль, 940

win32::ODBC модуль, 940
Win32::OLE модуль, 940
Win32::OLE::Const модуль, 941
Win32::OLE::Enum модуль, 941
Win32::OLE::NLS модуль, 941
Win32::OLE::Variant модуль, 941
Win32::PerfLib модуль, 941
Win32::Pipe модуль, 484
Win32::Process модуль, 941
Win32::Semaphore модуль, 941
Win32::Service модуль, 941
Win32::Sound модуль, 941
Win32::TieRegistry модуль, 448, 941
Win32API::File модуль, 941
Win32API::Net модуль, 941
Win32API::Registry модуль, 941
World Wide Web, 935
wrapper, 1064
write функция, 280, 282, 777, 893
  $% и, 722
  $- и, 722
  $= и, 722
  $| и, 729
  $^A и, 778
  $^a и, 716
  formline и, 778
  -i ключ и, 546
  $^L и, 722
  select и, 843
  длина строки и, 458
  нижние колонтитулы и, 286
  пример использования, 777
  строки шаблона и, 777

X

-x (extract program) ключ командной строки, 538
-x (доступен для выполнения с реальными uid/gid) оператор проверки файла, 135
-x (доступен для выполнения с текущими uid/gid) оператор проверки файла, 135
-x (извлечь программу) ключ командной строки, 554
/x модификатор шаблона, 188, 189, 192, 194
-X (отключение вывода предупредительных сообщений) ключ командной строки, 554
x (повторение) оператор, 55, 132
x= (присваивание повторения) оператор, 144
xor логический оператор, 59, 139, 147
XS язык
  внешние подпрограммы-оболочки, 584
  код, переносимость, 683
  средства отладки для, 936
  функции, соответствие между C и Perl, 585
XSLoader модуль, 939
XSUB, 585
  ввод и вывод, 589
  подпрограммы-оболочки для Perl, 584
xsubpp компилятор, 585

Y

y/// (транслитерация) оператор, 184, 197, 894
yield функция (Thread модуль), 505

Z

^Z (Control-Z) как EOF, 674
-z (нулевой размер) оператор проверки файла, 135

А

аббревиатуры, 931
абзацы, режим, 724
абстрагирование, 335
абстракция в объектно-ориентированном программировании, 358
автодекрементирования (- -) оператор, 128
  магический, 402
автозагрузка, 1050
  генерация методов доступа с помощью, 385
  методов, 376
автоинкрементирования (++) оператор, 128, 403, 1050
  магический, 128
автоматическая
  генерация, перегрузка, 1050
  обработка конца строки, 548
  очистка буферов, 729, 1053
аддитивные операторы, 132

адреса
имена сокетов как, 744
повторно используемые, вывод содержимого, 573
получение из имен хостов, 783
сетевые
получение из имен хостов, 781
трансляция в имена, 780, 783
упакованные сокетов, 787
язык C, в сравнении с Perl, 644
адреса в памяти
для байтов, 1066
левые значения для, 1061
адреса оператор (в C), 148, 1066
алгоритмы, 1050
анонимные, 1050
каналы, 477
массивы
срез двумерного массива, 319
формирователь, 292
хеш, создание, 323
объекты ссылки, 290
подпрограммы, 262
присвоение имен на этапе выполнения, 342
формирователь, 294
хеши
добавление к многомерным хешам, 327
массив анонимных хешей, формирование, 325
структуры данных для объектов, 365
формирователь, 292
аппаратные устройства, 1080
аргументы, 1050
командной строки, 1050
программы, в оболочках и Perl, 645
фактические, 1081
формальные, 717, 1081
арифметические операторы, 129, 1051
бинарные арифметические операторы, 54, 129
перегрузка, 397, 400
порядок вычислений, 55
арность, 123
архитектура, 1051
асинхронные события, 1051
ассоциативные массивы, 757
ассоциативность операторов, 123, 138, 1051
атомы, 244, 1051
в поиске по шаблону, 245
атомарная операция, 626
атрибуты, 1051
attributes прагма, 936
locked и method, использование с потоками, 391
locked, подпрограмм, 510
lvalue, подпрограмм, 390
в объектно-ориентированном программировании, 367
интерполяция переменных и, 818
класса, 806
сохранение состояния для, 392
объекты, хеши атрибутов, 45
подкласс, замещение для надкласса, 381
подпрограммы, имена, синтаксис, 277
файлы, операторы проверки, 60
аутентификация, модули Perl для, 936

Б

байты, 82, 1051
адреса в памяти, 1066
операторы поразрядного сдвига, 133
символы и, 452, 739
чтение, 831, 874
байт-код, 582, 1051
байтовые значения, сокращенные обозначения классов символов, 209
базы данных
связывание с переменными типа хеш, 412
соединения Perl с, 179
управление базами данных, 1046
базовые классы, 356, 937, 1051
вызов всех замещенных методов, 375
для фильтров и трансляторов pod, 937
замещение методов, 372
конструкторы и, 368
предоставление для всех классов модулем UNIVERSAL, 937
базовые модули, Perl, 601
базовое время ($^T), 718
бесконечные циклы, 156
библиотеки, 1052
C/C++, загрузка как расширений Perl, 938, 939

ExtUtils::Liblist модуль, 938
Perl, установка модулей CPAN в, 605
библиотечные файлы Perl, поиск для, 555
математическая библиотека C, 590
не установленные расширения из MakeMaker, 938
определение, 896
бинарные операторы, 54, 123, 124, 1052
перегрузка, 396
бит исполнения, 1052
битовые маски, 844
блоки, 63, 84, 150, 151, 160, 299
continue блок, 154
голые, 162
объявления пакетов, область охвата, 338
хеши, 1052
циклы и, 796
блокирование, 1052
блокировка сигналов, 468
блокировка файлов, 469
DBM-файлы, 473, 757
flock функция, 774
снятие блокировки при закрытии файла, 471
блокировки с совместным и монопольным доступом, 470
блокирующие вызовы, блокировки в потоках, 507
блоков буферизация, 729
больше (>) оператор, 138
больше или равно (>=) оператор, 138
булев, 1053
контекст, 107, 1060
bool оператор преобразования, 400
m// (поиск) оператор в, 191
булевы
значения, 38
операторы, 61
определение истинности, 62
буфер команд в отладчике, 558
буферизация, 1053
IPC::Open2 в, 967
seek с, 841
блоков, 729
блочная, 1052
ввод одного символа и, 779
двунаправленные каналы, проблемы с, 482
команд, 729, 1053
пример использования, 491
неочищенные буферы в версиях Perl, 776
получение односимвольного ввода, 971
при использовании pipe, 826
с помощью exec, 768
строк, 729, 1053

В

вариадические функции в Perl, 265
ввод, 774
XSUB, 589
дескрипторы файлов, 52
использование ioctl для, 793
каналы, 477
двунаправленные, 482
многозвенные конвейеры, 478
тринаправленные, 482
одного символа, буферизованный и небуферизованный, 779
операторы, 116
проверка указателей файлов для, 844
разделитель записей, 729
файлы, нахождение конца с помощью eof, 548
функции для, 740
вектор битов, предоставление интерфейса массива для, 448
векторы, строки как, 862, 889
величины неопределенные, значение истинности, 62
верблюды
розовые и голубые, 705
сравнение с лошадьми, 35
версии, Perl, 728
модулей, проверка, 352
определение с помощью $^V, 730
отличия Perl 4 и 5, 645
указание, 543
в строке #!, 539
верхний регистр, преобразование в, 97, 193
взаимоблокировка в потоках, 508
взломщик, 1053
взятие по модулю оператор (%), 131
виртуальные каналы (как потоки TCP), 489

виртуальная машина, Perl, 524, 526
висячие команды, 1053
владение файлом, 751
вложенные
массивы, 316
подпрограммы, 309
списки, 316
структуры данных, 44, 563
массивы и хеши, ссылки на (в массивах), 40
внешние подпрограммы, переносимость, 683
внутренние имена переменных, Perl, 90
возведение в степень целых чисел, 595
возведения в степень (**) оператор, 129
возврат, 203, 223
каретки, 96, 678
в классе символов POSIX, 218
символ, 209, 246
возможные соответствия, задание множества, 230
восьмеричные числа, 95, 808
задание $/, 543
символы, представление, 206, 453
временные значения, 86
временные файлы, 960
File::Temp и, 628
POSIX модуль и, 970
риск использования для системы безопасности, 626
создание с указателем файла, 963
время, 879
Time::Local модуль, 931
для местного часового пояса, 800
обращения к файлу/модификации файла, 863, 888
по Гринвичу (GMT), 789
программа UDP для получения с других машин, 495
создание пауз, 851
существования файла, 137
функции Perl для, 741
встраивание Perl, 582, 591, 938, 1054
выполнение команд Perl из C, 594
добавление интерпретатора Perl в программу C, 592
компиляция встроенных программ, 592
стек Perl, обработка из C, 595
встроенные
методы, вызов замещенных, 372
встроенные
документы, 102, 1054
конец файла, 1060
отступы, 103
точка с запятой, 640
типы данных, 85
функции, 1054
замещение, 353
перечисление по типам, 937
символьная семантика, действие с, 455
создание прототипов подпрограмм для эмуляции, 272
встроенное уничтожение, 379
входные файлы, режимы открытия, 811
вывод, 52, 774
XSUB, 589
выбор указателей файлов для, 843
вложенных структур данных, 563
данных через системный вызов низкого уровня, 828
использование ioctl для, 793
каналы, 477
STDOUT как канал в другую программу, 478
двунаправленные, 482
фильтрация, 480
массивов и хешей, изменение стиля (в отладчике), 573
массивов хешей, 326
многомерных массивов, 318, 323
многомерных хешей, 327, 328, 331
накопитель форматированного вывода, 778
проверка указателей файлов для, 844
разделители полей и записей, 729
содержимого typeglob, 573
текущий выбранный канал, 1078
форматы записей, объявление, 280
функции для, 740
хешей массивов, 324
выделение заглавными буквами
escape-последовательности для, 97
uc и ucfirst функции, 881
в именах модулей, 931
верхний регистр, преобразование в (управляющий символ), 193
голые слова, проблемы с, 100
изменение, 199

имен
методов, связанных переменных, 413
меток, 154
подпрограмм, 264
пакетов/модулей, 660
вызывающий, 1054
вызов, 461
по значению, 265, 1054
по ссылке, 265, 269, 1054
подпрограмм, косвенный, 263
функций, 126
выполнение
автоматическое (анализ опций), 573
других программ из Perl, 875
кода, 524
условное, 59
методов, 358
программ Perl, 50, 517, 537
выравнивание, 281
по левому краю, 285
по правому краю, 285
по центру, 285
выражения, 1054
expr и list, 151
goto и, 166
s///e и, 252
циклы и, 155
высокомерие, 1054
выход из
бесконечных циклов, 156
блоков if или unless, 162
однократно выполняемых блоков, 162
отладчика Perl, 567, 572
программ, 34
выходные файлы, режим открытия, 811
вычитание
из наборов символов, 216
модификаторов шаблонов из замыканий, 230
перегружаемых объектов, 397

Г

генераторы Perl, 672
других языков на Perl, 673
фильтры исходного кода, 675
генераторы кода, 528, 939, 1055
Bytecode генератор, 528
на C, 529
генерируемые шаблоны, 252
глаголы, 47
пакеты, обработка в, 46
гласные звуки, классификация слоговых азбук соответственно, 214
глобальный поиск, 192
глобальные переменные, 169, 1055
local, использование с, 175
локализация, 798
объявления, 166, 173, 817
переменные пакета в качестве, 337
потоки, доступ из, 505
предварительное объявление прагмой vars, 936
управление использованием с помощью прагмы strict, 178
глобализация, 82
голые
блоки, 162
использование в структурах case, 163
идентификаторы, 101
(простые) слова, 100, 1055
strict прагма, проверка использования, 178
гравис, 117
границы (слов, строк), 222, 223
группы
ID, 751
имена, 780
пользователей, 1055
имя группы, 780
функции для, 741
члены, имена регистрации, 779
процессов, 465, 784, 847
возврат приоритета, 784
убиение, 796
функции для, 740
группы имен, Exporter, 353
группировка операторов в выражениях, 125
группировки оператор, 182, 201
групповые метазнаки, 206, 457
классы символов и, 208, 209

Д

дампы памяти, 1055
dump функция для, 763
Thread::Signal модуль, предотвращение, 514

запуск сигналами в библиотеке C, 464
проблемы отказа в обслуживании при поиске по шаблону, 637
данные
байт-ориентированные и символ-ориентированные, 452
ненадежные, 610
доступ к командам и файлам при ограниченных правах, 619
код, маскирующийся под данные, 635
обнаружение и отмывание меченых данных, 613
окружение, зачистка, 618
обеспечение экранного дампа, 936
фиксированной длины, функции для, 740
данные экземпляра, 367, 380, 1068
генерация методов доступа с помощью автозагрузки, 385
использование замыканий для закрытых объектов, 387
новые функции, Perl 5.6, 390
порождение классов с помощью Class::Struct, 384
создание методов доступа с помощью замыканий, 387
создание отдельных функций для доступа, 381
даты, переносимость, 684
датаграммы, 489, 1055
двоичный, 1055
режим для указателей файлов, 744
двоичные
данные, pack функция, 819
файлы, 815
в качестве обычных файлов, 61
декомпиляция с помощью модуля B::Deparse, 532
операторы, 139
двумерные массивы, создание и доступ, 316
двунаправленный
вывод, свойства Unicode, 213
канал, открытие с помощью связанного указателя файла, 441
двунаправленная связь, каналы, 482
действия (отладчика)
вывод полного списка, 563
выполнение команд, задание из отладчика, 565
декларативный подход, программирование регулярных выражений, 252
декомпозиция символов в более простые, 213
декомпрессия модулей CPAN, 604
декремента (- -) оператор, 58
декрементирование переменных, 58, 1055
деление, 131
демоны, режим проверки меченых данных, важность активации, 611
деревья грамматического разбора реконструкция, 517
деревья синтаксического разбора, 516, 520
внутренние, сохранение, 527
исходные, реконструкция и сращивание с текущим, 529
дескрипторы файлов, 36, 52, 773, 844, 1055
syslog, привязка выдачи к, 448
typeglob для, 115
анонимные, 295
ассоциированные форматы, имена по умолчанию, 280
возвращение для указателей файлов, 438
имена, 89
локализация, 115
неявное создание ссылки в, 290
оканчивающиеся на _TOP, 282
оператор угловой скобки и, 117
передача, 272, 476
связывание, созидающие указатели файлов, 441
создание, 52
ссылки на, 295
указатели файлов, передача с помощью, 474
деструкторы, 378, 1056, 1072
сборка мусора с помощью, 379
совместно используемая память и семафор, 488
управление, 556
десятичные точки, выравнивание, 281
диагностические сообщения, 982
диаграммы, генерация, 280

диапазоны
наборы символов, задание для, 198, 208
операторы (.. и ...), 141
динамическая компоновка, 586
с исходным кодом C из Perl, 584
динамическая область видимости, 168, 169, 171, 1056
local, использование с глобальными переменными, 175
переменные шаблонов, 187, 193
переменные шаблонов,
директивы (pod), 689, 1056
дисциплины, 815
указателей файлов, установка с помощью binmode, 744
длина
найденной строки, 221, 726
строк, 797
Unicode, 456, 458
в шаблонах, 650
добавление
членов в существующий хеш, 326
элементов в массивы, 318, 885
документация
perl, страницы руководства, 21
библиотечные функции C, 738
встраиваемая в программы Perl, 686
доклады об ошибках, 28
к стандартным модулям, 346
книги по Perl, 25
модули для, 937
системное средства просмотра по умолчанию, вызов, 568
документирование программ, 84
домашняя страница Perl, 27
дописывание строки к строке с помощью .=, 57, 144
дополнение элементов в массивы, 830
доступ к
замещенным методам, 372
записям сложной структуры данных, 331
массивам хешей, 326
многомерным хешам, 328
срезам многомерных массивов, 319
хешам массивов, 324
элементам связанных массивов, 423
доступ к данным, потоки, 505
блокировка подпрограмм, 509
взаимоблокировка, 508
методы блокировки, 510
переменные условий, 511
разблокирование, 508
синхронизация с помощью lock, 506
доступа методы, 367
доступность для записи, оператор проверки файла, 61, 135
доступность для чтения, оператор проверки файла, 61, 135
дочерние процессы, 890
ID, возврат, 776
запуск для записи в, 477
читаемые, запуск, 478
драйверы аппаратного обеспечения, модули Perl для взаимодействия с, 601

Е

единицы компиляции, 92, 1056
естественные языки, 34
переменные окружения, управляющие их обработкой в Perl, 555

Ж

жадный поиск, 71, 220, 221, 244, 1056
лексемы, 84
жесткие ссылки, 127, 289, 291, 1056
%SIG массив, 463
использование, 297
замыкания, 306
на имена файлов, 798
жизненный цикл программ на Perl, 516
реконструкция дерева грамматического разбора, 517
фаза выполнения, 517
фаза компиляции, 516

З

завершение
escape-последовательностей, модифицирующих символы, 97
бесконечных циклов, 156
медленных операций по тайм-ауту, 467
процессов, 795
сигналы, обработчик для, 463
заглавный регистр
и ucfirst функция, 199, 458
заглавные буквы, 89
в именах модулей, 348

заглушки для модулей, создание, 938
заглядывание, 247
вперед, 247, 1056
назад, 247, 1056
заголовки, имена форматов, 286
заголовочные файлы, 1056
загрузка структур данных с диска, 332
загрязнение пространства имен, 350
задание темы, 45
закрытые методы, игнорирование наследования с помощью, 377
закрытые объекты, использование замыканий для, 387
закрытые переменные, 804
закрытия при exec флаг
socket и, 851
socketpair и, 852
закрытость, 356
модули Perl и, 349
замыкания, 172, 306, 308, 867, 1056, 1057
в потоках, 501, 502
в шаблоне, 229
вложенные подпрограммы, эмуляция с помощью, 309
закрытые объекты, использование для, 387
как шаблоны функций, 308
пример использования, 378
присвоение glob для определения подпрограмм, 344
создание, 263
методов доступа с помощью, 387
замена
элементов массивов, 856
найденных подстрок, 193
замещение
методов базовых классов, 372
функций, 353
замещенные методы (базового класса), вызов всех, 375
запись
в сегменты совместно используемой памяти, 849
в файлы, режим открытия и, 811
данных через системный вызов низкого уровня, 829, 876, 893
записи
переменной длины, 282
разделитель ввода, 729
различных структур данных, 331
генерирование хеша из, 332
формирование, доступ и вывод, 331
функции для, 740
запуск, 52
зарезервированные слова, 89
голые слова, возможность конфликта с, 100
захват в шаблонах, 226, 1057
подавление в кластеризации, 228
слов, буквенно-цифровых, 192
защита данных, 17, 609, 863
Win32::FileSecurity модуль, 940
взлом, 1053
модули для, 936
обработка ненадежных данных, 610
доступ к командам и файлам при ограниченных правах, 619
окружение, зачистка, 618
обработка ошибок синхронизации, 622
временные файлы, 626
ошибки защиты в ядре Unix, 623
ситуации гонки, 624
операторы тестирования файлов и, 913
работа с ненадежным кодом, 629
защищенные разделы для, 630
код, маскирующийся под данные, 635
защищенные разделы для ненадежного кода, 630
знаки
метазнаки, общие групповые , 206
пунктуации, поиск границ слова и, 223
значения
изменение во время копирования, 1063
массивы, 101
ограничение области видимости, 170
по умолчанию, 1057
скаляры, 94
сравнение с переменными, 36
ссылки, 290
установление неопределенными, 882
хешей, сортировка, 795
зомби процессы, 1057
уборка, ветвление серверов и, 494

И

И (&&) логический оператор, 59 140
И (&) поразрядный оператор, 139
идентификаторы, 83, 90, 786, 1057
  GID (ID группы), 1046
  PID (ID процесса), 465, 815
    угадывание, 627
  в сравнении с именами, 90
  голые слова и, 101
  заключение в фигурные кавычки, 98
  модулей Perl, 347
  одинарные кавычки (') в, 96
  пакетов, 338
    в таблице имен пакета, 339
  подчеркивание (_) в качестве первого символа, изменения между версиями Perl, 645
  потоков, 504
  сегментов совместно используемой памяти, 849
  семафоров, 846
  содержащие буквенно-цифровые символы Unicode, 456
  текущие uid и gid в сравнении с реальными, 610
    защита от, 619
идеограммы (идеографы), 70
  поиск с помощью, 457
идеографические унарные операторы, 129
идиомы Perl, 662
избыточность в Perl, 60
извлечение подстрок, 868
ИЛИ оператор, 59, 233
имена, 88
  typeglob, 115
  атрибутов, 277
  в сравнении с идентификаторами, 90
  временных файлов, 627
  групп, 780
  запись в хеши, 39
  импортирование в текущий пакет, 347
  классов, различение с подпрограммами, 364
  команд, 47, 1057
  компьютеров, 47
  конструкторов, 367
  массивы, 88
имена
  метки и дескрипторы файлов, 89
  методов, связанных переменных, 413
  модулей/пакетов, 347, 660
    использование заглавных букв, 348
  ограничение области видимости, 170
  операционных систем, 728
  отсутствие в списке экспорта или удаление из списка импорта, 351
  пакетов, 339
  параметры, отсутствие необходимости имен, 265
  переменных, 36, 87, 660
  подпрограмм, 89, 263
    выделение заглавными буквами, 264
  поиск, 91
  полностью квалифицированные, 90
  пользователей, 786
  портов, получение по номеру, 786
  программ, 731
    исполняющегося Perl, 721
    сценария Perl, 731
  протоколов
    получение из номеров портов, 787
    трансляция в/из чисел, 784
  путей, для модулей, устанавливаемых в Windows, 347
  регистрации, 779
    getgrent функция, 779
    getgrgid функция, возврат, 780
    getgrnam функция, возврат, 780
    getlogin функция, 782
  сетевые адреса, получение из, 783
  соглашения Perl по использованию заглавных букв, 89
  сокетов, 744
  таблица имен, 340
  трансляция в сетевые адреса, 783
  файлов, 104, 1057
    & (амперсанд) в, 813
    globbing, 120
    внешние данные, риск для системы безопасности, 618
    изменение, 836
    поиск по шаблону, 120, 682
    получение из символических ссылок, 833

имена
файлов
просмотр другой программы или команды eval, 565
расширение, 788
режимы открытия, 811
связывание жесткими ссылками, 798
связывание символически, 869
хеш %INC, содержащий, 723
форматы, ассоциированные с дескрипторами файлов, 280
формата, 777
функций, 47
хеши, 88
хостов
Sys::Hostname модуль, 934
получение из сетевых адресов, 780
трансляция в адреса, 781, 783
экспорт, 349, 352
именованные
каналы, 484, 1057
символы, 96
вставка, 456
метазнаки для, 206
унарные операторы, 133
императивный стиль программирования, 251
импорт, 347, 350
имен из одного пакета в другой, 341
подпрограмм из другого модуля, 263
семантики в пакет, 886
инвертирование массивов и хешей, 839
инвокант, 358, 1054
индексы, 649
в массивах, 40, 324, 716
отрицательные, 85
индексирование, 1058
инициализация
выражения, 155
объектов, 367
отладчика, настройка через файлы инициализации, 570
переменных перед вызовом подпрограмм, 269
переменных типа хеш, 114
срезов массивов и хешей, 132
инкапсуляция, 44, 301, 356, 1058
на основе пространств имен, 448
объектов в замыканиях, 388
инкремента (++) оператор, 58
интерполирующий контекст, 108
интерполяция
значений, являющихся массивами, 101
переменных, 37, 97, 1058
glob оператор, использование, 122
в строках, заключенных в двойные кавычки, 55, 97, 183
поиск по шаблону, управление с помощью, 234
условных операторов в шаблоны, 260
списочных значений, 110
ссылки на найденный текст, 74
функции и, 426
шаблона на этапе поиска, 259
интерпретаторы, 36
Perl, 50, 524, 526, 1058
встраивание в приложения, 527
доступ из C, 591, 592
вызов посредством строки #! (shebang), 538
взаимодействие с компиляторами, 532
запуск нескольких, 527
команд, 537
кавычки в системах Unix, 541
потоки, 498
уничтожение объектов при завершении работы, 379
интерфейсы, 356, 1058
сравнение байтовой и символьной семантики в, 454
инфиксные операторы, 54, 127, 1058
иррегулярные файлы, 61
исключающего ИЛИ (xor) оператор, 139, 147
исключительные ситуации, 1058
die и, 760
в отделенных дочерних потоках, 504
возбуждение для индикации ошибок в подпрограммах, 267
возбуждение при неуспехе, 936
модули для, 937
ненадежные данные, 613
перехват, 733, 766
потоки, перехват, 503
причина фатальных ошибок, 1081
функции, возбуждающие их, 741

искусственные языки, 34
исполняемые файлы, 1058
истинные значения, 1058
вычисление в операторах if и unless, 63
история Perl, 703
исходный каталог, 748, 1058
исходный код
CPAN, для дистрибутивов Perl, 600
код, полученный извне, выполнение, 610
окно вокруг контрольной точки, изучение, 559
средства для разработки, 530
фильтры для, 529, 601, 675
итеративные операторы, 64
for циклы, 66
foreach оператор, 66
until оператор, 64
while циклы, 64
перегрузка, 403
итерация, 159, 1059

К

кавычки, 37, 311
в аргументах formlin, 778
в системах, отличных от Unix, 541
в фигурных скобках, 98
двойные, 183
интерполирующий контекст, 108
ложные значения, 1062
операторы кавычек, 98
пропуск, 100
строки, 39
изменение стиля вывода строк, 573
каналы
io::Pipe модуль, 933
двунаправленные
межпроцессное взаимодействие, 482
открытие с помощью связанного указателя файла, 441
закрытие, 753
именованные, 484, 1057
при межпроцессном взаимодействии, 477
общение процесса с собой, 479
указатели только для чтения, открытие, 812
канонизация, 1059
символов, сведения о, 457
каноническая декомпозиция, символов, 213
карантинная проверка подозрительного кода, 630
каталоги
DirHandle модуль, 933
io::Dir модуль, 933
opendir функция, 817
rewinddir функция, 839
seekdir функция, 842
закрытие, 754
изменение текущего, 748
корневой, переопределение, 752
оператор проверки файла, 61
присоединение спереди к @inc, 548
проверка файлов в деревьях, 932
создание, 802
создание и удаление переносимым способом, 932
текущий рабочий, получение пути к, 932
удаление, 840, 883
указатели, 1079
функции для, 740
чтение записей, 832
категории
символов, 212
функции, 739
квантификаторы, 70, 71, 182, 201, 219, 650
атомы и, 244
жадные, 221
классы символов и, 209
минимальные и максимальные, 220, 202
точка (.) метасимвол, использование с, 207
классы, 336, 355, 1059, 1062
Perl, 208
Thread класс, 501
базовые, 372
в сравнении с модулями, 337
данные, управление в, 391
хранение ссылок на данные класса в самом объекте, 394
наследование, 369
base прагма, 371, 936
автозагрузка методов, 376
замещенные методы, доступ к, 372

игнорирование с помощью закрытых методов, 377
класс UNIVERSAL и, 374
номер версии, возвращение, 375
объектно-ориентированные модули в качестве определения, 346
пакеты, 127, 358, 877, 886
порождение с помощью модуля Class::Struct, 384
производные, 1071
реализация
в виде псевдохешей, 806
связанных указателей файлов, 433
связанных хешей, 427
связывания, 412
связывание скаляров, 415
функции для, 740
цитирование имени пакета для, 363
классы символов, 70, 208, 1059
групповые метазнаки и, 206
инвертирование, или отрицание, 208
метазнаки, используемые в, 204, 209
поиск свойств символов в Unicode, 457
пользовательские, 208
предопределенные, доступность, 457
путаница с индексами массивов, 101
классы-контейнеры, содержащие указатели на структуры данных с ссылками на себя, 379
кластеризация
без захвата, основания для, 228
в шаблонах, 228, 1056
клиенты, 1059
TCP, 491
UDP, 495
получение имен клиентов, 493
ключевые слова, 1059
ключ/значение пары, 1082
ключи командной строки, 538, 543
- - ключ, 543
-0 ключ, 539, 543
-a ключ, 544, 721
-c ключ, 544
-d ключ, 544, 557, 578
-e ключ, 50, 538, 546
-f ключ, 544, 546
-h ключ, 546
-i switch, 548
-i ключ, 539, 546
-l ключ, 548
-m ключ, 549
-n ключ, 550
-P ключ, 550
-p ключ, 550
-S ключ, 551
-s ключ, 551
-T ключ, 494, 552
-t ключ, 610
-U ключ, 552
-u ключ, 552
-V ключ, 553
-v ключ, 553
-w ключ, 51, 553, 982
-X ключ, 554
-x ключ, 538, 554
односимвольные, обработка при слитном написании, 932
установка ключей, принимаемых по умолчанию, 555
ключи, хеши,псевдохеши, 42, 383
код
извлечение и отображение с помощью отладчика, 564
ненадежный, 629
защищенные разделы для, 630
код, маскирующийся под данные, 635
примеры использования модуля Safe, 633
повторное использование, 662
смешивание символьной и байтовой семантики, 455
эффективность, 655
коды операций, 520, 524, 583
команды
Perl, выполнение из C, 594
буферизация, 729, 1053
в pod, 689
висячие, 1053
доступ при ограниченных правах, 619
имена команд, 47
метки для, 1062
обработка, 537
местонахождение Perl, 542

команды
отладчика, 560
вывод структур данных, 563
выход из отладчика, 567
действия и, 565
контрольные точки, 562
нахождение кода, 564
перезапуск отладчика, 567
пошаговой обработки и прогона, 561
просмотр документации, 568
трассировки, 563
управление опциями, 568
повторный вызов, опция отладчика, 571
управления циклом for, 159
хеши функций, хранение в, 330
командная строка
вызов интерпретатора Perl из, 50
вид на экране, 572
модули для обработки, 932
отладчика, 572
комбинированная символьная последовательность, поиск с помощью, 207, 457
комментарии, 1060
многострочные, 687
определение, 84
отбрасывание перед обработкой, 233
расширение посредством модификатора шаблона /x, 189
синтаксис Perl в сравнении с C, 644
компиляторы, 1060
глобальные объявления и, 166
регулярных выражений, 239
компиляторы, Perl, 718
взаимодействие с интерпретаторами, 532
модули, относящиеся к, 601, 939
серверы для, 527
советы, 723
компиляция, 515, 518, 582
встроенных программ Perl (в C), 592
жизненный цикл программ на Perl, 516
реконструкция дерева грамматического разбора, 517
фаза компиляции, 516
модулей CPAN, 605
порядок, 536
конвейеры (|), 52
open функция, риск для защиты данных, 618
многозвенные, 477, 478
конец строки
поиск, 222
автоматическая обработка, 548
конкатенации (.) оператор, 55, 132
пример использования, 57, 132
самогенерация через обработчик преобразования в строку, 401
строковые константы, перегрузка, 408
эффективность, 652
конкатенация строк, 1060
оптимизация в компиляторе, 522
с помощью join, 794
константы, 342
Perl, загрузка определений fcntl.h как, 934
System V IPC, определение для, 934
перегрузки, 407
подстановка функций-констант, 274
конструкторы, 294, 355, 364, 1060, 1081
bless функция и, 365
new метод, 807
имена, 367
имена классов или объекты, работа с, 366
инициализаторы, 367
классов связанных переменных, 412
копий, 405
наследуемые, 365
проверка доступа, установка в, 389
контекст, 105, 1060
булев контекст, 107
интерполирующий (двойных кавычек) контекст, 108
пустой, 108, 891
скалярный и списочный, 105
контракт между модулем и пользователем, 349
контрольные
выражения, 563, 1060
суммы, 885
точки, 1060
вывод полного списка, 563
задание, 557, 560
контрольные выражения и, 563
удаление, 563
установка, 448, 562
конфигурация Perl, для отладки, 545

концевые пробелы при сравнении строк, 138
концевые символы строки в программах для Интернета, 494
копирование файлов, 932
копирования при записи семантика, 498
корневой каталог, переопределение, 752
короткого действия (short-circuit) операторы, 58
косвенная адресация, 288
косвенность между использованием и реализацией класса, 358
косвенный указатель файла, 809, 1061
косвенные объекты, 361, 1061
  неопределенность в, 361
  определение, 361
  пример использования, 332
  явный путь, передача в позицию, 613
косвенные ссылки, 93
криптография, 863
кэширование, 313, 485, 603
  классов символов, 217
  модули, обеспечивающие, 447
  объектов, хранимых в памяти, 392
  поиска метода, 371
  пример использования, 430, 663
  с помощью stat, 715
  файлов, 653, 780, 933

Л

левые значения (lvalues), 86, 88, 1047
  ?: и, 144
  lvalue, атрибут подпрограмм, 278
  операторы и, 56
  операторы поиска по шаблону, использование с, 186
  определение с помощью ref, 835
  подпрограммы и, 89
  присваивания операторы и, 145
  ссылки и, 127
лексемы, 83, 519
  выделение, 583
  проблемы с двусмысленными символами, 134
  разбор текста в список, 931
лексическая область видимости, 171, 1061
  закрытость и, 388
  замыкания и, 307
  объявления, 168
  объявления our и, 817
  пакетные переменные, поиск, 92
  переменные и, 268
    привязка переменных к, 91
  прагмы и, 177
  пространства имен и, 895
  сравнение с лексическими анализаторами, 519
  файлы и, 93
лексический анализ, 583, 1061
  в компиляторе Perl, 519
лексический анализатор, 1061
лексические переменные, 169, 283, 804, 1061
  вывод с помощью B::Showlex, 940
  доступ из функций, 268
  запись в стеки временной памяти при рекурсии, 526
  потоки, доступ из, 505
  с файловой областью видимости, хранение анонимных подпрограмм в, 377
  символические ссылки и, 311
  сохранение значений между вызовами функций, 268
  типизованные, 1079
лексические предупреждения, 177, 937
  ${^WARNING_BITS}, 735
  $^w и, 553
лень, 1061
липкий (sticky) бит , 654
листинг кода, 565
литералы, 180, 1061
  UTF-8, активация использования в, 456
  псевдолитералы, 116
  строки, 94
  числовые, 94
ловушка при отвязывании, 444
логические операторы, 58, 147, 1062
  (!) отрицания оператор, 129
  or (||) оператор, использование со списковыми операторами, 167
  И (&&) оператор, 140
  ИЛИ (||) оператор, 233, 140
  перегрузка, 401
  приоритет, 362
ложные значения, 1062

М

магические автодекрементирование и автоинкрементирование, 128, 142, 402, 419
максимальное и минимальное число соответствий, задание, 71
максимальное соответствие, 201, 220
маршрут для поиска библиотек, 723
массивы, 36, 39, 85, 87, 101, 1062, 1063
  @_ массивы, 265
    копирование значений в список my, 265
  AV (значение типа массив) определение типа в C, соответствие, 583
  exists, 770
  pop функция и, 826
  s/// (подстановка) оператор и, 196
  анонимные, объекты, реализованные как, 382
    use fields прагма, решение проблем с помощью, 382
  анонимные, ссылки на, 292
  ассоциативные, 757
  вложенные, 316
    двумерные массивы, создание и доступ, 316
  добавление элементов, 830, 885
  замена/удаление элементов, 856
  имена, 88
  инициализация с помощью оператора x, 132
  меченые, 612
  многомерные, 44, 316
  обработка, функции для, 740
  отрицательные индексы, отсчет от конца массива, 647
  первый элемент, 716
  передача по ссылке в функции и из них, 269
  переменные массивов, 101
  производительность и, 653
  разбор текста в, 931
  разделитель индексов, 115
  размер, 112
  связанные, функция delete и, 647
  связывание, 421
    Tie::Array модуль, 937
    методы для, 422
    удобство обозначения, 426
  списки и, 108
массивы
  ссылки на, 290
  стиль вывода, изменение в отладчике, 573
  удаление элементов, 848, 856
  хешей, 325
    генерирование, 325
    доступ и вывод, 326
  хеши массивов, 323
  элементы, получение временных значений, 799
массива контекст, 1060
математика
  библиотека (язык C), 590
  математические функции, перегрузка, 403
  операторы, 54, 123
межпроцессное взаимодействие, 461
меньше (<) оператор, 138
меньше или равно (<=) оператор, 138
метазнаки, 181, 200, 1062
  в шаблонах, 199, 203, 206
  синтаксис расширения, 202
метасимволы, 181, 654, 1062
  в аргументах exec, 768
  в шаблонах, 199
  оболочек, в командах каналов, 812
  обход с помощью quotemeta, 830
  обход с помощью обратной косой черты (\), 200
  структурирующие, 182
метки, 151, 159, 1062
  goto и, 165, 790, 996
  имена, 89
  пакеты и, 338
  циклы и, 154, 159, 660, 1013, 1059
методы, 301, 355, 1062
  Thread класс, 501
  UNIVERSAL класс, добавление в, 376
  автозагрузка, 376
  блокировки (методы объекта), 510
  вызываемые связанными переменными, 413
  вызов, 358
    подпрограмм в сравнении с, 366
    с использованием оператора стрелки, 359
    с помощью косвенных объектов, 361
    явный и неявный, 359

методы
выполнение как методов доступа для данных класса, 393
для связанных массивов, 421
для связанных указателей файлов, 433
для связанных хешей, 427
добавление к существующему классу, 373
закрытые методы, игнорирование наследования с помощью, 377
замещение, 372
имена, 661
класс UNIVERSAL, 374
наследование классами Perl, 380
объявления, указывающие на возврат l-значений, 390
подпрограммы в сравнении с, 359
подпрограммы как, 358
поиск
наследование классов, 370
наследование через массив @ISA, 370
ссылки на, 308
методы связывания
массивов, 422
скаляров, 415
хешей, 428
методы доступа, 1056
выполнение методов данных класса как, 393
генерация с помощью автозагрузки, 385
псевдохеши и, 383
создание, 380
с помощью замыканий, 387
методы классов, 355
имена пакетов в качестве инвокантов для, 358
конструкоры как, 365
методы экземпляров, 355
ссылки на объекты как инвокантов для, 358
минимальное соответствие, 72, 201, 220
минус (-) оператор, 129
многомерные
массивы, 44, 316, 1063
хеши, 327
генерирование, 327
доступ и вывод, 328
многоточие (...), 200
многоязычность, 685
модули Perl для, 933
текстовых данных, с помощью Unicode, 451
множества, пересечение, 270
множественные значения, 39
множественное наследование, 369, 1063
SUPER псевдопакет, работа с, 373
методы, вызов всех замещенных базового класса, 375
подпрограмма AUTOLOAD и, 377
модификаторы
операторы в сравнении с квантификаторами, 183
операторов простых, 150
шаблонов, 188
m// (поиск) оператор, 191
s/// (подстановка) оператор, 194
tr/// (транслитерация) оператор, 198
замкнутые, 229
модификации время, файла, 863, 888
модули, 16, 45, 336, 344, 346
CPAN, 46, 601
декомпрессия и распаковка, 604
категории, 601
компиляция, 605
обеспечение доступности программам, 604
отправка электронной почты, 683
переносимость, 684
создание, 606
сохранение в, 336
установка в библиотеку Perl, 605
безопасные потоки и, 500
в сравнении с классами, 337
встроенные функции, замещение, 353
для начальной загрузки, 938
для связывания в CPAN, 447
имена, 660
импортирование, 938
классы, хранение в модулях для закрытости, 377
потоков, 512
предотвращение экспорта имен, 352
применение заглавных букв, 90
расщепление в отдельные файлы, 345
серверы, 528
система классов и, 371

модули
создание, 349
закрытость и Exporter, 349
проверка версий, 352
стандартные, 930
DBM-файлы, загрузка управляющих библиотек, 935
Perl компилятор и генератор кода, 939
базовые классы и удобства, 937
для защиты данных, 936
многоязычность и национальные установки, 933
интерфейсы с операционной системой, исправление, 934
объектно-ориентированный доступ к файлам, каталогам и IPC, 933
обработка командной строки, 932
поддержка , 937, 938, 939
пользовательские интерфейсы, обеспечение, 935
предупреждения и исключительные ситуации, 937
расширения языка Perl, 936
сетевое и межпроцессное взаимодействие, 934
текст, работа с, 931
типы данных, 931
файлы, методы для доступа между платформами, 932
тестирование, 586, 590, 607
функции для, 740
экспортирование имен в другие модули, 792
монопольная блокировка файла, получение, 471
мультиобработка, 497
доступ к процессу, 622
модель потоков, 499
мультипликативные операторы, 55, 131
мутаторы, копирование и, 405

Н

наборы символов
ASCII, 685
Unicode, 451
переносимость, 685
упорядочение и диапазоны символов, 198
надклассы, 356
наследование, 356, 1063
базового класса, установка во время компиляции, 936
классов, 337, 369
одиночное и множественное, 369
конструкторы, 365
методов доступа к данным класса, 393
модулей, классу Exporter, 350
ограниченный доступ и, 390
перегрузка и, 409
реализация псевдохешей и, 384
через массив @ISA, 370
настройка отладчика, 569
файлы инициализации, 570
опции, 571
поддержка в редакторе, 569
функции для, 577
национальные настройки
locale прагма с классами символов POSIX, 218
Unicode и, 797
десятичные точки в форматах, 281
классы символов в Perl и, 209
модули и прагмы для, 933
объявления, при сравнении строк, 138
начальная загрузка модулей, 588
начальная позиция найденной строки, 726
начало страницы, обработка, 723
начало строки, поиск, 222
НДКА (недетерминированный конечный автомат), 241
(не граница слова) утверждение, 223
не равно (!=) оператор, 138
(не соответствует свойству), 457
небезопасные операции, 552
неинициализированные переменные, 38
неинтерактивный режим, перевод отладчика в, 574
неквалифицированные имена, поиск, 91
ненадежные данные, ошибки зависимости от, 613
неопределенные подпрограммы, вызов с помощью автозагрузки, 344
неопределенные скаляры, 94
неравенство, 138

(не-слово) метасимвол, 204
неупорядоченные списки, 41
неявные создание ссылки и разыменование, 290
неявные вызовы методов, 359
нижний регистр, преобразование в, 97
нижние колонтитулы, 286
номера
  портов, получение имен протоколов из, 787
  протоколов, 785
  страниц, 722
  строк, 104, 1063
    _ _LINE_ _ маркер, 674
    __line__ лексема, 104
    входного файла, 674, 724
нулевые
  значения, 1064
  списки, 110, 113
нулевой ширины утверждения, 181, 201
  квантификаторы и, 183
нумерованные переменные, 715

О

области видимости, 36, 88, 1064
  my и our переменные, 337
  блоков, 151, 171
  в операторах if, 153
  в потоках, 513
  в циклах for, 155
  в циклах foreach, 157
  в циклах while, 154
  динамическая, 175, 1056
  закрытые переменные, 804
  лексическая, 88, 91, 171
  обработка сигналов, 464
  объявления, 168, 169
  объявления пакетов, 337
  операции поиска по шаблону, 187
  прагма strict и, 47
  подпрограммы, 267
  стек области видимости, 525
  файл, 171
  функции для, 740
области памяти, левые значения, 56
оболочки, 1064
  C, вокруг сценариев set-id, 623
  Perl в сравнении с, 644
  альтернативные для Perl, системы Windows, 556
  Борна оболочка (sh), 540
оболочки
  кавычки, передача в, 546
  команды в, 1059
  команды каналов с метасимволами
    обработка, 477
    передача в, 812
  команды отладчика и, 567
  обход при открытии канала с несколькими аргументами, 477
  обход с помощью -|, 480
  опасности для защиты данных при использовании, 635
  переменные окружения, 554, 916
    временная установка, 554
    угрозы безопасности, 618
  потоки ввода/вывода, настройка, 52
  предотвращение использования, 620
  символы для порождения, 571
обработка
  верхнего колонтитула, 843,893
  команд оболочки, 538
  начала страницы, 282
  опций, модули Perl для, 932
обработчики, 1064
  перегрузка, 396, 397
    as_string, 400
    отсутствие, 406
  сигналов, 463
    определение как строк, 339
образчики (swatches), 217
обратные вызовы, 867, 1065
  в Pod::Parser, 699
  в XML::Parser, 670
обратные ссылки, создание с помощью скобок, 227
обратная совместимость, 1065
обращение команд print и printf (связанные дескрипторы файлов), 433
обращения к файлу, время, 863, 888
обрезание строки, 749
обычные файлы, проверка, 61
объекты, 44, 355, 1079
  ввода-вывода с перемещаемой позицией, 933
  данные экземпляров
    генерация методов доступа с помощью автозагрузки, 385
    закрытые объекты, использование замыканий для, 387
    новые функции, Perl 5.6, 390

объекты
порождение классов с помощью Class::Struct, 384
создание методов доступа с помощью замыканий, 387
управление, 380, 382
деструкторы экземпляров, 378
сборка мусора с помощью, 379
классы, 336
управление данными класса, 391
конструкторы, 294, 364
инициализаторы, 367
наследуемые, 365
косвенные, 1061
методы, 301
блокировки, 510
потоки, 501
ссылки на, 308
на основе связанных переменных, 412
наследование, 369
через массив @ISA, 370
освящение, 746
преобразование в строку, 305
помещение в другие объекты в сравнении с наследованием, 379
связывание ссылки с объектом, 127
связанные переменные в основе
разрыв ссылки на, 446
система объектов Perl, 357
ссылки на, в скалярах, 38
ссылок, 289, 357
уничтожение (глобальное), 556
функции для, 740
объектно-ориентированное программирование, 336, 355
абстракция в, 358
модули для, 346, 933
переносимые операции с файлами, 932
ссылки в Perl, эмуляция, 337
объявления, 149, 1065
our, 817
package, 824
structs, 384
глобальных переменных, 166
с лексической областью видимости, 173
методы, объявления, указывающие на возврат l-значений, 390
пакетов, 45, 169, 337
переменных, 149, 169
объявления
подпрограмм, 167, 263, 867
анонимных, 262
поля (прагмой use fields), 382
с областью видимости, 168
форматов, 280
ограниченный массив, создание, 422
ограничители, 1065
операторы поиска по шаблону и, 186
одиночное наследование, 369
однозначные переменные, 37
однострочники, 50, 1065
окончание, 795
бесконечных циклов, 156
округление чисел с помощью функции sprintf, 792
окружение, 1065
зачистка, 618
переменные, 719
октеты, 453
ООП, 355
операторы, 54, 123, 149, 1066
if и unless, 153
ассоциативность, 123
ввода, 116, 117
глаголы как, 48
глобальные объявления и, 166
группирования, вложенность, 227
доступ, ограничение в модуле Safe, 632
кавычки, синтаксическое сходство с, 39
логические, 58
перегружаемые, 125, 395, 398, 931, 1067
арифметические, 400
итеративный, 403
логического отрицания, 401
поразрядные, 401
преобразования, 399
присваивания, 402
разыменования, 404
сравнения, 403
циклические ссылки, проблемы с, 405
позиции и длина в Unicode, 458
поиска и подстановки, 180
поиска по шаблону, 183
модификаторы для, 188
приоритеты, 55, 123
в C, сохранение в Perl, 125

операторы
преобразования
булев контекст, интерпретация объекта в, 400
в число (не числа), 400
присваивания, 56, 646
простые, 150
скалярный контекст, передача параметрам, 38
составные, 151
сравнение с операторами управления циклом, 161
сравнения, 60, 138
неассоциативность, 138
строковые, 55
угловых скобок, 642, 546
argv и, 717
использование функции glob вместо него, 121
управления циклом for, 159
циклы, 153
языка C, отсутствующие в Perl, 148
операции, контекст и, 105
операционные системы, 13, 1066
#! прием, моделирование на системах не-Unix, 540
библиотеки поддержки потоков, 500
вызов интерпретаторов строкой #!, 538
имена модулей, трансляция в имена каталогов, 347
интерпретатор Perl, запуск, 50
интерпретаторы команд для, 537
интерфейсы, исправление, 934
модули Perl для, 601
надежность сигналов в, 467
основанные на GUI, 682
перевод строки, разновидности, 678
переносимость программ Perl, 600, 677
поддержка flock в, 470
поддержка fork, 477
сигналы и режимы по умолчанию, 464
сокеты, поддержка, 490
определения, 1066
классов (объектно-ориентированные модули в качестве), 346
подпрограмм, 264
загрузка из других файлов, 168
загрузка с помощью AUTOLOAD, 344
сравнение с объявлениями, 167
типов (язык C), 583
оптимизация
в компиляторе Perl, 519
производительность и, 648
опции, 188
освобождение памяти, 313
основание времени ($^T), 718
основное имя, файла, 932
остроконечники, 680, 821, 1066
пример, 823
открытие
сокетов, 851
файлов через системные вызовы низкого уровня, 871
отладка, 544, 557, 612
DB модуль, 936
Devel::Peek модуль для программ XS, 936
Perl, использование ключа -ddebugging компилятора C, 370
автоматическое выполнение отладчика, 573
включение в связанных дескрипторах файлов, 441
глобальное уничтожение объектов и других ссылок, управление, 556
для программистов C, 643
для программистов оболочки, 644
для раскаивающихся программистов Perl 4, 645
код отладчика, команда для загрузки, 555
команды отладчика Perl, 560
многомерные массивы и, 320
модули серверов для, 528
настройка отладчика, 569
перегрузки, 410
поддержка отладчика в Perl, 575
символическая, 1074
сообщения об ошибках, 28
управление из программ, 560
уровень отладки, установка в классе или экземплярах, 393
флаги, 719
частые ошибки новичков, 639
отображение ключей, в псевдохешах, 383

отрицание, 129
арифметическое, оператор (-) , 129
логическое, оператор (!) , 129, 401
классов символов, 72, 208, 209
POSIX, 218
поразрядное, оператор (~), 129
отрицательные индексы массива, 85, 647
отрицательное заглядывание вперед, утверждение, 244, 248
отсутствие соответствия, продолжение поиска, 224
отчеты, генерация, 280
охватывающий оператор, 1067
очереди, 315
Thread::Queue модуль, 512
очистка буфера, 729, 1067
очистка хешей, 431
ошибки, 774
$@ переменная, 720
$? переменная, 718
$! переменная, 728
Carp модуль, 943
CGI::Carp модуль, обработка, 944
warn функция, создание предупредительных сообщений, 892
в Perl, сообщение о, 28, 983
в подпрограммах, 267
запись в httpd или другие журналы ошибок CGI, 935
ненадежные данные, 613
неудачный экспорт имен, 352
нехватки памяти, 727
номер (errno), 1063

П

пакеты, 45, 88, 91, 335, 344, 1067
bless функция, аргументы, 364
в массиве @ISA, наследование через, 370
верхнего уровня, 92
добавление новых методов, 374
идентификаторы, 338
имена, 101, 660
импорт семантики в, 886
методы и подпрограммы, разрешение для, 359
объявление, 169, 337
ограничители для, 338
освящение объектов в, 746
по умолчанию, 169
пакеты
привязка переменных к, 877
расщепление для автозагрузки, 938
связывание переменных с, 886
таблицы имен, 340
текущий пакет, 1078
цитирование имени, 363
пакетные переменные, 91, 92, 169
потоки, отсутствие защиты в, 506
память, 1067
ID сегмента совместно используемой памяти, 849
вывод статистики использования, 556
нехватка, 727
сборка мусора, 313
совместно используемая, System V IPC, 485
управляемый доступ к, 486
управление, деструкторы и, 378
эффективность, 654
параметры, 717
подпрограммы, изменение по месту, 266
работа с, 265
скалярный контекст, создание для, 38
пароли, 863
crypt функция и, 755
passwd файл, 785
паузы в сценариях, 851
последовательные порты, модули Perl для работы с, 601
перебор с возвратом, 241
в поиске по шаблону, 241
подшаблоны без возврата, 249
перевод на верхний регистр
функции lc и lcfirst, 797
перевод строки, 33, 83, 678
в именах файлов, угроза защите данных, 619
в строковых литералах, 97
пустые строки, истинность значения и, 65
строки, содержащие его, поиск в них, 188
точка (.) метасимвол, соответствие, 207
перегруженные строковые константы, трансляция текста с помощью, 261

перегрузка, 125, 395, 1067
nomethod и fallback (переход в аварийный режим), 406
overload прагма, 396, 931
Overloaded (метод в прагме overload), 409
диагностика, 410
константы, 407
конструктор копий (=), 405
математических функций, 403
на этапе исполнения, 410
наследование, 409
обработчики, 397
определение, в применении к встроенным сущностям Perl, 106
перегружаемые операторы, 398
преобразования в строку, 439
самогенерации, 401
сортировка и, 403
функции для, 409
циклические ссылки, как избежать, 405
передача
по значению, 1054
копирование значений @_ в список my, 265
по ссылке, 265, 1054
прототипы объявлений функций для, 290
ссылок, 269
перезапись существующих файлов, ситуация гонок и, 625
переключение режима трассировки (в отладчике), 563
переменные, 85, 87, 126, 288, 297
$[, символы, 716
B::Xref модуль, перекрестные ссылки на C, 531
local, 643
my, 305
our, 337
Perl в сравнении с C, имена, 644
анонимные, 291
атрибуты, получение и установка, 936
вывод дампа командой V отладчика, 341
глобальные, 1055
закрытые, 804
значения, присваивание и, 57
имена, 36, 660
переменные
имена во вложенных таблицах имен, 91
инициализация перед вызовом подпрограмм, 269
интерполяция, 37
класса, 391
контрольные точки, помещение на, 448
лексические области видимости, 804
локализация, 798
меченые, 612
модули, экспорт в программы, 350
неинициализированные, 38
объявление, 169, 149
глобальных, 173
лексическими, 171
окружения, 554, 1068, 719
Env модуль, 936
PERLDB_OPTS, установка опций отладчика с помощью, 574
perl_destruct_level, 379
в оболочках и Perl, 645
использование Perl модулей, 556
переносимость, 682
проверка чистоты данных, 611
отладчик, настройка, 570
пакетов, 91, 337
вывод, 564
использование памяти, 573
объявления, 817
помещение объявления в область видимости блока, 393
хранение данных класса в, 392
переменные условий, 511
повторные подстановки, выполнение, 196
полностью квалифицированные, доступ из наружного пакета, 631
программирование оболочки, в сравнении с Perl, 644
пунктуации, обработка модулем English, 936
сброс значений, 838
связывание, 411, 877
ловушка при отвязывании, 444
массивы, 421
с пакетами, 886
скаляры, 413
указателей файлов, 433
хешей, 427

переменные
скалярные, 114
специальные, 714
сравнение со значениями, 36
статические (в C/C++), 268
строки, поиск в известном наборе строк, 235
типа массив, 114
типа хеш, 114
трансляция с помощью, 199
управляющие, в цикле foreach, 156
условий, 511
установление области видимости, использование прагмы strict, 47
форматирования, 284
переменной длины записи, 282
переменной длины символы, 453
перенос текста, 287, 931
переносимость
gethostent и, 782
проверка эффективности, 656
программ на Perl, 677
сведения о, 462
сигналов, 682
файлов и файловых систем, 681
функций, информация о, 742
пересылка данных, модули для, 948
пересечение множеств, 270
перехват
выдачи
внешней команды, 875
функций, 481
исключительных ситуаций, 766
ошибок при нехватке памяти, 727
сигналов, 464
обработчик для, 463
управления
в неявных циклах, 550
у объектов во время уничтожения, 378
планирования и вытеснения политика, потоки, 500
повторения (x) оператор, 55, 132
повторно используемые адреса, вывод содержимого, 573
повторное использование кода, 662
повторяющиеся замененные символы, удаление, 198
подача листа, 722
поддержка разработки, модули для, 601
подклассы, 356
атрибуты, замещение для надкласса, 381
подпрограммы, 37, 48, 262, 358
B::Xref модуль, перекрестные ссылки на C, 531
do SUBROUTINE оператор, 763
Perl, вызов из C, 593
автозагрузка, 344, 376, 938
анонимные, 262
ссылки на, 294
аргументы, 717
атрибуты, получение и установка, 936
блокировка для потоков, 509
locked атрибут, 510
вложенные, 309
внешние, 584
переносимость, 683
возврат ссылок, 294
возвращаемые значения, 265, 838
возвращающие l-значения, 390
вопросы областей видимости, 267
вызов
методов в сравнении с, 366
выполнение без пошагового прохода, 561
косвенный, 263
выполнение, 1054
данные, передача в и из, 264
замена с помощью goto, 790
имена, 88, 89, 263
именованные, взаимодействие компилятор/интерпретатор, 532
импорт из другого модуля, 263
классы с такими же именами, 363
команда отладчика для перечисления по шаблону, 565
константные, 342
методы в сравнении с, 359
объявление, 167, 867
в пакете с AUTOLOAD, 377
определение, 264
определения, загрузка из других файлов, 168
ошибки, индикация, 267
передача ссылок в, 269
переопределение и подстановка, 275
подстановка с помощью goto, 166
прерывание с помощью exit, 771
производительность и, 650

подпрограммы
простые слова, путаница с, 645
прототипирование, 271
осторожное использование, 276
разрешение метода, шаги для, 370
с лексической областью видимости, 804
списки параметров, работа с, 265
ссылки
на извлечение или передача, 341
в скалярах, 38
стек вызовов, 746
управление циклом и, 162
подпространства имен для переменных, 89
подстановка
вычисление при, 254
оболочки в сравнении с Perl, 644
оператор, 108
постоянных функций, предотвращение, 275
функций-констант, 274
подстроки, 1069
замена с помощью оператора s///, 193
подшаблоны
без возврата, 249
кода в регулярных выражениях, 255, 1069
пожирание (slurping) файлов, 543, 1048
позиции, 222
substr функция, манипуляция с помощью, 222
длина строки и, 458
поиск границы слова, 223
начала и конца строки, 222
приоритет при поиске по шаблону, 242
поиск
grep функция для, 791
и замена, глобальные, 194
линейный, в сравнении с хешами, 648
маршрут для поиска библиотек, 723
одного байта в языке C, групповой метазнак, 206, 457
оператор (??), 838
по шаблону, 244
(m//) оператор, 801
с возвращением, 1069
имен файлов, переносимость, 682
поиск
файлов по шаблону имени, 120, 932
подстрок, 193, 792, 839
правила поиска имен, 92
регулярные выражения при, 68
сценариев, 551
пользовательские интерфейсы, модули для обеспечения, 935
пользователи
аутентификация, модули Perl для, 603
время использования CPU, 880
группы, 1055
извлечение информации о, 741
предоставление ограниченных прав, 610
полиморфизм, 356, 1069
положительные утверждения с заглядыванием, 244, 248
поразрядные операторы, 139, 147
И (&) оператор, пример использования, 971
отрицания (~) оператор, 129
пример использования, 971
перегрузка, 401
поразрядного сдвига операторы, 133, 144, 1069
порты, 1069
трансляция имя/номер, 786
портированные приложения Perl, в операционные системы, 600
последовательное соответствие, 243
последовательности символов, вводимых с клавиатуры, генерация сигналов с помощью, 463
постоянный интерпретатор Perl, 591
постоянные структуры данных, предоставление через tie, 447
постфиксные операторы инкремента и декремента, 58, 124
потоки, 497, 800
locked и method атрибуты, 391
блокирование, 800
множественные, в одном интерпретаторе, 527
модель потоков, 499
доступ к данным, 505
управление, передача между потоками, 512
модель процессов, 498

потоки интерпретаторов, 498
Perl API для, 527
пошаговое выполнение кода, 561
поэзия Perl, 706
правые значения (rvalues), 86
права доступа, 872, 881
операторы прав доступа к файлам, интерпретация, 136
пользователи, предоставление ограниченных, 610
прагмы, 47, 168, 176, 336, 549, 886, 900
charnames, 96
имена, применение заглавных букв, 90
модули и, 348
с лексической областью видимости, 177
предварительная обработка Perl, 674
предварительное объявление подпрограмм, 377, 936
предварительное расширение массивов, 653
предупреждения, 553, 892, 982, 1070
${^WARNING_BITS}, 735
-w ключ, включение с помощью, 51
модули для, 937
о неопределенных величинах, 149
о ссылках на связанные объекты, 446
отключение постоянное, 554
перехват, 732
форсирование словесных, 936
преобразование
в строки, 399
модуль для, 948
перегрузка, 439
ссылок, 318, 320
структур данных, 334
восьмеричных чисел в десятичные, 808
времени, 789, 800
десятичных чисел в двоичные, восьмеричные или шестнадцатеричные, 860
значения ASCII в символ, 752
значения Unicode в символ, 752
между целыми числами и символами UTF-8, 458
между языками, 675
символов в значения ASCII, 817
списковых значений в строки, 794, 819
преобразование
строк в списковые значения, 857, 884
типов данных между C и Perl, 595
чисел из/в шестнадцатеричные, 195, 791
препроцессор (язык C), 550, 1070
препроцессинг Perl, 1070
прерывания сигналы, обработчик для, 463
префиксные операторы, 54, 124
инкремента и декремента, 58
метасимволы в качестве, 200
приведение типов данных, 94, 1070
приведения типа оператор (в C), 148
привязка, 73, 313, 1070
операторы (=~, !~), 69, 184
переменных к пакетам, 411, 877
файла DBM к хешу, 756
приоритеты, 126, 737, 1070
defined, 124
my и, 170
диагностических сообщений, 982
для подпрограмм, возвращающих левые значения, 278
И (and) относительно &&, ИЛИ (or) относительно ||, 141
изменения между версиями Perl, 646
логических операторов, 59, 147, 362
модификаторов относительно запятых, 170
освещение в электронной документации, 22
поиск с использованием регулярных выражений, 241
процессов, 784, 847
сверху вниз, список операторов, 124
унарных операторов (выше бинарных), 133
присваивание
условному оператору ?, 144
операторы, 56, 86, 144, 1070
возвращаемые значения, 57
перегрузка, 402, 405
связанных переменных, 418
спискам, 111
элементам двумерных массивов, 317
элементу связанного массива, 423
присоединение спереди, каталогов @inc, 548

пробельные символы, 70, 83, 658, 737, 1071
/x модификатор поиска по шаблону, 189
в форматах, 282
ведущие, удаление во встроенных документах, 103
использование в качестве имени переменной или как разделитель в конструкции кавычек, изменен, 646
канонизация в переменных, 196
символы кавычек, использование в, 99
проблемы отказа в обслуживании (шаблоны и регулярные выражения), соображения безопасности, 637
проверка доступа, установка в конструкторе, 389
проверка меченых данных, 494, 613, 552, 609, 610, 611, 1062
-d и, 554
File::Glob и, 958
PERL5LIB и, 555
активация с помощью -T, 611
защищенные разделы и, 633
интерполированный код и, 637
отмена для указателей файлов, 964
программы CGI и, 633
программы, требующие, 610
режим проверки, 610
функции, возвращающие меченые данные, 741
шаблоны и, 258, 918
проверка файлов, 60, 625, 135, 1066
программы, 262
выполнение, 875, 1054
выход, 34, 771
дампы памяти в, 763
однострочники, 1065
пошаговое выполнение с помощью отладчика, 561
пример расчета и вывод, 49
управление отладчиком, 560
программирование, Perl, 639, 662
генераторы программ, 672
других языков на Perl, 673
фильтры исходного кода, 675
частые ошибки новичков, 639
переносимость, 677
программы манипуляции с файлами, создание переносимых, 681
стиль, 251, 658
эффективность, 648
программное обеспечение с открытым исходным кодом, 1071
проекты (большие), преимущества использования прагмы strict, 47
производительность, 16
измерение, 880
профилирование, 578
производные классы, 356, 1071
методы как оболочки методов базового класса, 372
тест пустого подкласса, 1079
пространства имен, 88, 336, 1071
для пакетов, 45
доступ, ограничение с помощью модуля Safe, 630
инкапсуляция, основанная на, 448
модули и, 349
простые (голые) слова
изменения между версиями Perl, 645
причины избегать, 642
протоколы, трансляция имя/число, 784
прототипы, 271, 1071
& (амперсанд), пропуск в именах подпрограмм, 271
встроенные функции, эмуляция, 272
для замыканий, 309
неявная передача по ссылке в, 290
осторожность при использовании, 276
подстановка функций-констант, 274
профайлер, Perl, 578
профилирование с помощью Devel::DProf, 936
процедуры, 47
процедурный стиль программирования, 251
процессы
fork функция, 776
STDIN, STDOUT и STDERR в процессах, 52
umask для, 881
завершение, 795
зомби, 466
идентификаторы (PID), 815

процессы
межпроцессное взаимодействие, 461, 469
System V IPC, 485
в Интернете, 461
на одной машине, 462
сокеты, 489
многозадачные среды, доступ к CPU, 622
ожидание, 890
передача сигналов группе, 465
переменные для, 720
получение группы для, 847
потоки и, 498
приоритет, возврат, 784
приоритет, установка, 847
проверка существования, 465
функции для, 740
зомби, уборка, 466
псевдокоманды, конвейера, 479
псевдолитералы, 116
псевдонимы, 1071
в циклах for/foreach, 157
поиск и замена элементов в массивах, 196
для команд отладчика, 568
записей таблицы имен, 91, 116
символов, 205
создание присвоением typeglob, 341
псевдооператоры, перегрузка и, 399
псевдохеши, 302
use fields прагма, использование с, 383
моделирование с помощью функции overload, 404
реализация классов с помощью, 806
наследование и, 384
пустые строки, истинность значения, 65
пустой
контекст, 58, 108, 112, 660, 891, 1060
оператор угловых скобок (), 53
подкласс, проверка, 369

Р

равенства операторы, 138
разыменовывающие символы, 411, 1072
разыменование, 290, 297, 298, 299, 1072
typeglob, 340
адреса оператор (*), C язык, 148
значений хеша как функций, 330
операторы, 299, 404
элементов массива, 318
разделение процессов, 498
разделитель полей, задание нового, 544, 1069
разделители, определение с помощью оператора split, 68
размер массива, 112
разные функции, 740
разрушение файлов, 811
распаковка модулей CPAN, 604
расширение Perl, 584
XSUB и язык XS, создание оболочек для Perl, 585
ввод и вывод в XSUB, 589
использование функций из внешней библиотеки C, 590
создание расширений, 586, 936, 1072
расширение имени файла, 788
расширение массивов, 112
расширения языка, модули Perl для, 601
расширенные символы, API системные вызовы, использующие, 454
расшифрование, 755
реальный
GID ($REAL_GROUP_ID), 731
UID ($REAL_USER_ID), 719, 732
реальные значения, проверка с помощью функции defined, 757
ревизия, 730
регистр, 199
escape-последовательности для обработки, фаза интерполяции переменных, 205
операторы перевода, использование таблиц Unicode, 458
поиск без учета, 188
поиск с учетом, выключение, 188, 192, 194
поиск с учетом, отключение, 237
преобразование строк в верхний регистр, 199
регулярные выражения, 68, 179
редактирование по месту ($^I, $INPLACE_EDIT), 546, 724
редакторы, поддержка отладки, 569

реестр (Microsoft Windows)
.pl расширения, 541
обработка, 448
работа с, 941
режим автоматического расщепления (с ключом -a), 544, 1050
режимы, открытия файлов, 810
рекомпиляция шаблонов, ограничение с помощью модификатора /o, 189
рекурсия
в поиске по шаблону, 259
временных стеков лексических переменных, 526
подпрограмм, 267
реляционные базы данных, 448
связывание хешей, 447, 448
родительские классы, 356
родительские процессы, 776

С

самогенерация, перегрузка, 401
предотвращение, 406
самое левое наибольшее совпадение, 221, 1073
самооживление, 127, 300, 771, 1073
typeglob, 296, 435
указателей файлов, 809, 962
сборка мусора, 313
глобальных переменных, 1073
объекты, ассоциированные со связанными переменными, 418
с помощью методов DESTROY, 379
свойства
Unicode, 208, 214, 458, 459
в объектно-ориентированном программировании, 367
связывание
модули для связывания в CPAN, 447
операторы (=~, !~), 130
переменных, 411
ловушка при отвязывании, 444
массивы, 421
с пакетами, 886
скаляры, 413
указатели файлов, 433
хеши, 427
связанные функции, 412
связки модулей, 601, 1073
связующий язык, Perl как, 52, 1073
сдвиг влево (<<) оператор, 133
сдвиг вправо (>>) оператор, 133
семантика, импортирование в пакеты, 886
семафоры, 1074
IPC::Semaphore модуль, 934
System V IPC, 485
Thread::Semaphore модуль, 513
Win32::Semaphore модуль, 941
использование при блокировке файлов, 472
совместно используемая память, 486
функции для, 845
серверы, 492
TCP, 491
ветвление клонов для обработки входящих соединений, 493
компиляторов, 527
модули, 528
общий интерфейс для, 940
режим проверки меченых данных, важность активации, 611
сервисы Интернета, CPAN-модули для, 490
сериализация структур данных Perl, 936
сетевое взаимодействие
информация о сети, функции для извлечения, 741
клиенты, 491
модули для, 490, 934
серверы, 492
сетевые адреса, трансляция в имена, 780, 783
сетевые модули Win32, 940
сетевые файловые системы, 1047
соединения клиент/сервер, 1076
сигналы, 461, 462
%SIG хеш, 462, 732
Perl в сравнении с языком C, 644
Thread::Signal модуль, 514
блокировка, 468
дампы памяти, запуск сигналами в библиотеке C, 464
завершение медленных операций по тайм-ауту, 467
завершение процессов с помощью, 795
источники, 463
передача группе процессов, 465
переносимость, 682
перехват с помощью прагмы sigtrap, 464, 934

потоки, доставка в, 500
процессы зомби и, 466
символы, 82, 246, 1074
байты и, 452, 739
в прототипах, 272
в форматах, 820
в шаблонах, 180
ввод одного символа, 779
восьмеричные, 95
вставки (^), 139
групповые символы для поиска, 206
декомпозиция, 213
длина в, 797
замена в строках, 197
метасимволы, 181
перевода строки, удаление, 54, 749
получение ASCII-значений, 817
получение из значений ASCII, 752
получение из значений Unicode, 752
преобразование между фиксированной 8-разрядной и переменной длиной в UTF-8, 453
псевдонимы для, 205
разделители, 83
разыменовывающие символы перед именами переменных, 411, 1072
регистр, 881
свойства, 216, 1073
задание собственных, 216
удаление, 749
шестнадцатеричные, 95
символьная семантика
в сравнении с байтовой, 454
действие, 455
символическая отладка, 1074
символические ссылки, 289, 310, 801, 869, 1074
lstat функция и, 801
опасность для защиты данных в Unix, 623
получение имен файлов из, 833
символическое разыменование, 178
проверка с помощью прагмы strict, 178
синтаксис, 1074
в Perl, 86
проверка, 544
синтаксический анализ, 583
в компиляторе Perl, 519
интерполяция в двойных кавычках и регулярные выражения, 183
синхронизация, 1075
доступ к данным в потоках, 506
ошибки, обработка, 622
временные файлы, 626
ошибки защиты ядра Unix, 623
ситуации гонки, обработка, 624
системные вызовы, 869, 1075
Perl, язык C в сравнении с, 644
перезапуск, 467
прерывание сигналами, 467
производительность и, 654
ситуации гонки
блокировка файлов и, 417, 470
медленные операции и, 467
обработка, 624
ядро Unix и, 623
скаляры, 36, 37, 85, 1075
Perl, преобразование в типы C, 594
SV (скалярное значение) в C, 583
жесткие ссылки как, 290
значения, 94
v-строковые литералы, 104
встроенные документы, 102
интерполяция значений, являющихся массивами, 101
операторы, 125
числовые литералы, 95
истинность, вычисление в, 62
константные, 342
методы связывания, 415
неупорядоченный набор, 41
операторы поиска по шаблону, применение к, 186
переменные
в контексте двойных кавычек, 194
в угловых скобках, 120
имена, 87
связывание, 413
определения базового класса, предоставление, 937
переменные волшебных счетчиков, 419
предотвращение использования нелокализованных переменных $_, 420
символы, длина в, 797
сложные структуры данных, представление в виде, 44
списки параметров и возвращаемых значений, подпрограммы, 264
списки, преобразование в, 43

списки скаляров, 39
ссылки, 38
функции для обработки, 739
скалярный контекст, 105, 1060
/g модификатор шаблона, использование в, 192
l-значения в, 170
m// (поиск) оператор в, 191
булевы значения и, 107, 1053
выражения в, 151
диапазона (..) оператор, 141
запятой оператор в, 146, 647
навязывание, 840
операции, действующие иначе в, 642
переменные типа хеш, вычисление в, 114
подпрограммы, вызов и выполнение в, 265
подтипы, 106
присваивание списку в, 112, 146
пустой контекст, 108
условный оператор в, 143
функции в, 738
скобки ( ), 43
для классов символов, 70
массивы и, 43
скорость, 578
слабые ссылки, 313
слова, 1075
захват, 192
списки, программируемое завершение команд по, 935
(слово) метасимвол, 70, 204
поиск идеографа с помощью, 457
сложение, 55
перегруженный оператор «плюс» (+), 397
сложные структуры данных
представление в виде скаляров, 44
создание в Perl, 315
случайные числа, 831, 863
генерация, 863
создание в виде указателей и скаляров, 444
смещения, 1075
read функция и, 831
seek функция, 841
результатов m//, 827
советы компилятору, 723
совместимая декомпозиция, символов, 213
совместно используемая память, System V IPC
управляемый доступ к, 486
функции для, 849
соединения, 1076
закрытие, 850
сокеты
listen функция для, 798
TCP и, 489
домена Unix, 489
закрытие, 753, 850
закрытие соединения, 850
модули для, 934
низкоуровневый доступ, функции для, 740
открытие с помощью функции socket, 851
отправка сообщений на, 846
перевод строки, отправка через, 678
получение сообщений на, 834
получение упакованного адреса сокета, 783
при межпроцессном взаимодействии, 489
передача сообщений, 495
сетевые клиенты и серверы, 491
прием соединений от клиентов, 742
прикрепление адреса к, 744
протоколы и, 462
соединение, 754
установка опций, 848
функции для, 490, 787
сообщения
IPC::Msg модуль, 934
System V IPC, 485
встраивание сценариев в, 554
вывод при входе в подпрограммы и выходе из них, 572
об ошибках, 662, 982
эффективность использования, повышение с помощью, 657
отправка на сокеты, 846
передача с помощью UDP, 495
получение на сокеты, 834
функции для, 803
сортировка
ключей и значений в хешах, 794
массивов, входящих в хеш, 324
перегрузка и, 403

списков, 852
хешей, 329
сортирующая последова-тельность, 1076
составные свойства Unicode, 211
состояние
наследование с помощью методов доступа к данным класса, 394
объекты класса и, 391
сохранение структур данных, 334
специальные переменные, 714
списки, 39, 75, 106, 108, 109, 151, 1076
map функция и, 801
аргументы функции как, 736
вложенные, 316
инвертирование, 839
интерполяция, 110
нулевые, 110, 1064
обработка, функции для, 740
преобразование в скаляры, 43
применение chop, 750
присваивание, 111
репликация с помощью оператора x, 132
скаляры, значения, возвращаемые процедурами, и их параметры, 264
слов, 111
сортировка, 852
сравнение с массивами, 108
текущих потоков процесса, 505
уплощение, 736
упорядоченные, записи данных в массивах, 322
списковые операторы, 124, 126, 147, 1076
унарные, их различение, 134
списочный контекст, 74, 105, 1060, 1076
foreach оператор, обеспечение, 66
l-значения в, 170
m// (поиск) оператор в, 191
m//g, перечисление всех найденных соответствий шаблону, 192
выражения в, 151
вычисление выражений в отладчике, 564
диапазона (..) оператор, 141
запятой оператор в, 146
навязывание скалярного контекста, 840
оператор угловых скобок в, 118
оператор обратной косой черты, предоставление, 305
операции, действующие иначе в, 642
переменные типа хеш, использование в, 114
подпрограммы, вызов и выполнение в, 265
присваивание списку в, 146
условный оператор в, 143
функции в, 738
хеши в, 42
сравнение
времени выполнения альтернативных версий кода, 939
объектов Thread, 505
строк, 138, 640
файлов, 932
сравнения операторы, 60, 138
sort функция и, 852
перегрузка, 403
срезы, 88
local и, 170, 176
инициализация с помощью оператора x, 132
интерполяция, 97, 727
массивов
пример использования, 111, 142
связанные с ними ошибки, 641
синтаксис, 128
многомерных массивов, 319
присвоение срезу, 105
ссылки и, 304
стрелки и разыменование, 300
хешей, пример использования, 911
ссылки, 38, 288, 833, 1076
HTTP, поиск, 69
Perl, сравнение с указателями C, 148
блокировка, 507
в многомерных массивах, 316
в таблицах имен на другие таблицы имен, 340
гиперссылки, 69
жесткие, 127, 1056
значение истинности, 62
косвенные, 93
косвенная адресация и, 288
на анонимные данные, 291
на дескрипторы файлов, 295
на записи таблицы имен, 342
на обработчики сигналов (пользовательские), 462

ссылки
на найденный текст, 74, 191, 226, 641, 715, 1076
доступ с помощью пронумерованных переменных, 226
шаблоны, порождаемые найденными подстроками, 259
на объекты, 294, 301, 357, 364
инвокантов для методов экземпляров, 358
перегрузка и, 395
связанных переменных, разрыв, 446
на подпрограммы, 264
извлечение или передача, 341
на связанные массивы, 422
на связанные скалярные переменные, 415
на структуры данных, 290
на указатели файлов, 809
на функции, хранение в структурах данных, 330
на хеши, 322
многомерные хеши, 331
на элементы таблицы символов, 296
объектно-ориентированное программирование и, 337
определение, 289
ослабление и разрыв, 379
передача, 269
по ссылке, 265
превращение в объекты, 365
предоставление как ключей хеша, 937
преобразование в другие типы указателей, 95
преобразование в строки, 305, 312, 318, 320
проверка, 835
символические, 310
слабые, 313
создание, 291
управление удалением, 556
фигурные и квадратные скобки, 311
циклические
использование памяти, 313
перегрузка, как избежать, 405
стандартные свойства Unicode, 212
стандартное устройство ввода/вывода, 52
статическая компоновка, 586
статическая область видимости, 171
статические переменные (в C/C++), 268
статус завершения процесса, 718
стеки, 41, 315
jumpenv, 526
Perl, обработка из C, 595
возврата, 526
вывод командой T, 559
вычисление выражений, 86
виртуальная машина Perl, 525
дисциплин, 816
контекста, 525
маркеров (mark stack), 526
операндов, 525
сохранения, 525
подпрограммы, текущие вызовы, 746
стиль программирования, 658
страницы руководства, 21, 1078
Pod::Man модуль, 937
подкаталог doc архива CPAN, 600
стрелка (->) оператор, 127
для разыменования ссылок, 299
строки, 36, 88, 281
eval функция и, 649
index функция и, 792
rindex функция и, 839
study функция и, 865
vec функция и, 889
буферизация, 729
в двойных кавычках
изменения между версиями Perl, 647
интерполяция и конкатенация, 55
интерполяция символов в, 96
вывод, 828
значений в, 306
восьмеричные, 808
двоичные, 808
единицы компиляции как, 92
заключенные в кавычки, 39
замены, создание с помощью модификатора /e, 254
значение истинности, 62
извлечение подстрок, 868
конкатенация, 132, 652
с помощью функции join, 794
контекст строк, 106
модификация, 195
отображение, изменение в отладчике, 573

строки
операторы, 55, 640
диапазона, работа с, 142
поразрядные, действие, 139
чтения, 53
первый символ, 716
перевод в верхний регистр, 797
поиск и замена подстрок (s/// оператор), 193
преобразование
в списковые значения, 884
в числа, 94
0x, 0b и 0 префиксы, обработка, 95
объектов в, 305
ссылок в, 305, 312
продолжения, 160, 722, 1071
разрыв, 722
расщепление на подстроки, 857
содержащие перевод строки, поиск в них, 188
соединение, 794
списковые значения, преобразование в, 819
сравнение, 60, 138
ссылки и, 312
утверждения границ, 222
форматы для, 860
шаблона, 777
шестнадцатеричные, 808
строковый контекст, 1060
строковые литералы
v-строки (вектор или версия), 104
перевод строки в, 97
строчки, 1078
строчно-ориентированная расстановка кавычек, 102
структуры, форматирование, 823
структуры данных, 315, 1078
для объектов, 365
записи более сложных, 331
массивы хешей, 325
многомерные хеши, 327
объекты, 355
организация в Perl, 322
плоские, линейные в Perl, 288
постоянные, предоставление через tie, 447
преобразование в строки, 334
сложные, представление в виде скаляров, 44
структуры данных
со ссылками на себя, классы-контейнеры, содержащие указатели на, 379
сохранение, 334
ссылки на, 290
хеши массивов, 323
хеши функций, 330
структурирующие метасимволы в регулярных выражениях, 182
структурные метазнаки, 200
суперпользователь, 1078
существительные, 35
пакеты, обработка в, 46
существование процесса, проверка, 465
сценарии
CGI, режим проверки меченых данных и, 611
вывод дампа памяти, 552
выполнение, 875
время выполнения, 137
встраивание в сообщения, 554
завершение с помощью exec, 768
имена, 731
интерпретация, оболочки в сравнении с Perl, 645
использование идиом Perl, 662
компиляция с ключом -e, 546
концевой маркер, 104
косвенная передача через стандартное устройство ввода, 538
написание, 546
ненадежные, 629
отладка, 544
поиск, 551
и компиляция, 538
проверка синтаксиса, 544
производительность, 648
связывание с интерпретатором Perl, 50
создание пауз, 851
стиль, 658
тестовые, 939
частые ошибки новичков, 639
счетчик строк, 753
счетчики, переменные волшебных счетчиков, 419

Т

тыква, 1079
таблицы, 316

таблицы имен, 88, 91, 340
в полностью квалифицированных именах, 90
пакеты, 339
работа с псевдонимами записей, 116
содержащие ссылки на другие таблицы имен, 340
создание временных значений оператором local, 176
текущий пакет, определение использования, 337
таблицы символов
Symbol модуль, 931
вывод для пакетов, 573
записи с именами символических ссылок, 290
идентификаторы имен форматов в, 778
пакеты, 825
ссылки на, 296
теги HTML и XML, преобразование текста в, 261
текст, 88
Unicode как интернациональный носитель для, 451
данные pod, преобразование в форматированный ASCII, 937
лексическая область видимости, 88
модули Perl для, 931
обработка, возможности Perl, 179
перенос, 287
текстовые редакторы
подсчет числа строк, различия с Perl, 83
сценарии, ввод, 50
текстовые файлы (-t оператор проверки файла), 61, 137
текущий
GID ($EFFECTIVE_GROUP_ID), 719, 731
UID ($EFFECTIVE_USER_ID), 719
пакет, 1078
телеконференции, 27
теневые пароли, 786
термы, 86, 1079
взаимоотношения, выражение с помощью операторов, 123
приоритеты, 126
терминал для ввода/вывода при отладке, 574
тест пустого подкласса, 1079
тернарный оператор, 54, 124
?: в качестве, 142, 164
отсутствие перегрузки, 397
тестирование модулей, 607, 608
тестирование файлов, -p, проверка FIFO, 484
типы данных, 36, 1079
typeglob, 115
внутренние, в Perl и C, 583
встроенные, 85
модули для, 931
объявления типов в Perl, 805
преобразование между C и Perl, 595
скаляры, 94
типизованные переменные
подтипы, преобразование, 94
скаляры, 94
точки привязки, 222
транслитерация, 184, 880
трансляция, 894
между языками программирования, 675
имя/номер портов сервисов, 786
трассировка в отладчике, 559, 563
тревоги (звонка), символ, 96
тригонометрия с помощью Math::Trig, 931
троичный оператор, 1079
тупоконечники, 680, 1079
упорядочение в pack, 819
упорядочение посредством vec, 889

У

уборка процессов зомби, 466
серверы, 494
угловых скобок оператор, 117
в связанных дескрипторах файлов, 435
удаление
всех действий отладчика, 566
каталогов, 840, 883
контрольных точек, 563
найденных, но не замененных символов, 198
переменных окружения, 720
символов, 749
файлов, 883
элементов массива, 826, 848, 856
элементов хеша, 759
косвенный, 1061

указатели, 85
в языке C, 288
каталогов, 933, 1079
на строки, 821
связанные, определения базового класса для, 937
ссылки, невозможность преобразования в, 95
указатели файлов, 1080
open функция, 808
SelectSaver модуль, 933
блокировки файлов по, 471
выбор для выдачи, 843
для каналов, явное закрытие, 812
дублирование, причины для, 814
использование методов объектов, 933
косвенные, 753
локализация, 799
надежность в сравнении с именами, 625
оканчивающиеся на _TOP, 723
передача через IPC, 473
через стандартный указатель файла, 474
позиционирование указателя в файле, 841
связывание, 433
методы для, 434
функции для, 740
чтение байтов данных из, 831
укорачивание массивов, 113
умножение, 55
строк, оператор повторения, 55
унарные операторы, 54, 123, 124
арифметические, 58
булевы, 61
идентификация по виду, 642
идеографические, 129
именованные, 122, 126, 133
перегрузка, 396
поведение как функций, 126
прототипирование функций как, 276
списковые операторы, их различение, 134
уничтожение потоков
detach метод, 504
join метод, 502
упакованный адрес сокета, 783
управление последовательностью выполнения команд, функции для, 740
управляющие
переменные, 156
символы, 96, 193
метазнаки в шаблонах, 206
не-ASCII на Macintosh, 542
структуры, 61
истинность, определение, 62
уровень подробности сообщений отладчика, 571
усечение, 1080
массивов, 113
списка аргументов, 573
чисел, 792
условия гонки, тесты файлов и, 913
условное выполнение кода, 59
условный контекст, 107
условные операторы, 153
(?:) оператор, 142, 164
выражения в циклах, 155
запись без фигурных скобок, 152
интерполяция в шаблоны, 260
установка
модулей CPAN, 604
в библиотеку Perl, 605
модулей, поддержка, 938
устранение неполадок, 557
устранение ошибок защиты Unix, 623
устройства (аппаратные), 1080
устройство ввода/вывода, выбор в отладчике, 574
утверждение позиции, 192
порядок соответствия, изменение с помощью, 231
утверждения (в шаблонах), 181, 201, 1080
заглядывающие, 247
классы символов и, 209
нулевой длины, 222
определение собственных, 261
позиции, 192, 222, 223
приоритет, поиск по шаблону, 244
учет времени
выполнение альтернативных вариантов кода, сравнение, 941
работы сценария, 137
сравнение при выполнении подпрограмм, 578

Ф

файлы, 1080
do FILE оператор, 762
open функция, 808
Win32API::File модуль, 941
владелец и группа, изменение, 751
возврат текущей позиции для, 877
временные, 960
время обращения к файлу и модификации, 863, 888
время существования, 137
дампов памяти, 552
двоичный режим, 744
дескрипторы, 52, 474
доступ при ограниченных правах, 619
закрытие, 753
запись через системный вызов низкого уровня, 876
изменение имен, 836
исполняемые, 1058
конец файла (EOF), 764, 1060
межпроцессном взаимодействии при, 469
механизмы блокировки, 469
передача указателей файлов, 473
модулей, автозагрузка, 345
модули Perl, размещение в отдельных файлах, 336
модули для доступа между платформами, 932
области видимости, 172
операторы проверки, 60
открытие
переносимым образом, 682
через системные вызовы низкого уровня, 871
переименование, программа для, 767
переменные с лексической областью видимости, 93
получение статистических сведений, 863
права доступа, 748, 872, 881
резервных копий, 548
различие между текстовыми и двоичными, 815
символическое связывание, 869
составляющие пути, разделитель для, 681
удаление, 883
усечение, 880
функции для, 740
чтение через системный вызов низкого уровня, 874
файловые системы, 1080
переносимость, 681
фаза выполнения, 517
сравнение с этапом выполнения, 518
фаза компиляции, 583, 1080
фатальные ошибки, 1058, 1081
фигурные скобки, 42, 63
фиксированные 8-разрядные символы, 453
фильтры, 1081
для исходного кода, 76, 675
фильтрация вывода при открытии с ветвлением, 480
флаг закрытия при exec, 629, 734, 743
fcntl и, 475, 772, 815
флаги, 188, 543, 1081
формальные аргументы, 717, 1081
форматы, 280, 1081
B::Xref модуль, перекрестные ссылки на C, 531
pack/unpack, 819
вывода накопитель, 778
лексические переменные в, 283
нижние колонтитулы, 286
низкоуровневый доступ к форматированию, 286
номера страниц, 722
объявление, 280, 777
обработка начала страницы, 282
переменные, 284
символы в, 820
списки аргументов для, вычисление в списковом контексте, 646
строк для, 860
целых чисел, 822
функций вызовы
pipe, низкого уровня для двунаправленной связи, 483
функции, 47, 262, 736, 1081
Perl, различия между платформами, 678
Perl, создание оболочек вокруг C, 585
return операторы в, 159
Thread модуль, 501
автозагрузка, 344, 376

функции
аргументы по умолчанию, как избежать ошибок, 642
байт-ориентированные оболочки для, 455
библиотека C, 738
в списковом или скалярном контексте, 738
возвращающие меченые данные, 741
встроенные, перечисление по типам, 937
генерирование, 307
замещение, 353
имена, 661
временных файлов, создание, 627
интерполяция, 426
модули, экспорт в программы, 350
настройка отладчика, 577
обработки сигналов, 463
обратного вызова, 307
операторы и, 123
именованные унарные, 133
определяемые пользователем, 262
открытые перегрузки, 409
отладчик, вызов внутренних, 570
относящиеся к сокетам, 490
перечисление по категориям, 739
ссылки в качестве входных и выходных данных, 270
хеши функций, 330
шаблоны для, 308

Х

хакер, 1081
хеши, 36, 39, 41, 85, 113, 1082
exists функция и, 770
HV (значение типа хеш) определение типа в C, 583
%sig, ссылки на обработчики сигналов, 462
анонимные
как структуры данных объектов, 365, 380
ссылки на, 292
блоки, 1052
вывод в отсортированном виде, 157
вместо линейного поиска, 648
возврат значений, 888
имена, 88
хеши
инвертирование, 839
инициализация с помощью оператора x, 132
каскадные условные операторы, использование вместо, 165
ключи, 42, 85, 1059
возврат с помощью функции each, 740, 764
возвращение списков ключей, 794
организация данных и доступ к ним, 322
поиск числа в, 115
предоставление ссылок как, 937
пути к модулям (в хеше %inc), 347
таблица имен, 340
удаление, 427
фигурные скобки ({ }), заключение в, 42
массивы хешей, 325
генерирование, 325
массивов, 323
генерирование, 323
доступ и вывод, 324
методы связывания, 428
меченые, 612
многомерные, 327
генерирование, 327
доступ и вывод, 328
обработка, функции для, 740
организация данных в Perl, 322
пары ключ/значение, 42, 85, 113
возврат с помощью функции each, 764
захват из строк оператором m//g, 192
передача по ссылке в функции и из них, 270
плоские таблицы доступа, обеспечение доступа к, 315
привязка файлов DBM к, 756
с ограниченным доступом, реализация, 390
связывание, 412, 427
Tie::Hash модуль, предоставление определений базового класса, 937
сложных записей, 332
ссылки как ключи, 312
ссылки на, 290

хеши
стиль вывода, изменение в отладчике, 573
таблицы имен, 340
typeglob в, 340
пары ключ/значение, 340
удаление значений из, 759
элементы, получение временных значений, 799
хосты, 935, 1082
хранение сложных данных в файле DBM, 447
хронометрирование, с помощью аварийных сигналов, 743

Ц

целые числа, 791, 822, 913, 931, 968, 1082
IV (внутреннее целочисленное значение), 583, 1047
Math::BigInt модуль, 395,
в C, 85
в ключах командной строки, 965
возведение в степень, 595
действие поразрядных операторов, 139
компактное хранение, 654
перегрузка, 407
плотная упаковка в строки, 819, 889
порядок байтов в, 680
преобразование в символы UTF-8, 458
стандартные модули для арифметики с, 931
строки как векторы из, 889
форматы, 95, 822, 860, 861
шаблоны, примеры использования в, 234
циклы, 64, 154, 641, 660, 1082
eval функция в, 649
for циклы, 66, 155
foreach цикл, 66
last оператор, 67, 156, 796
next оператор, 67, 807
redo оператор и, 834
reset функция и, 838
s/// (подстановка) оператор, использование с массивами, 196
while циклы, 64, 154
бесконечные, 156
циклы
извлечение срезов многомерных массивов, 319
использование в глобальных подстановках, 196
итерация, 1058
комментарии и пустые строки, отбрасывание перед обработкой, 233
метки для, 159, 1059
операторы управления, сравнение с, 161
проход (итерация) циклов, 155
управляющие команды в, 1059
утверждение, использование в, 225
циклические ссылки, 313
перегрузка, как избежать при, 405
разрыв, 379
цитирование имени пакета, 363
цифры, 70
в именах, 90

Ч

чередование, 182, 1050
в шаблонах, 230
классы символов и, 209
приоритет при поиске по шаблону, 243
соответствие одному или другому (...|...), 201
числа, 36
выравнивание по десятичной точке, 281
восьмеричные, 95, 808
запись в массивы, 39
запятые, вставка в, 285
значение истинности, 62
количество команд в буфере команд отладчика, 558
плавающей точкой с, 94, 822, 1068
Math::BigFloat модуль, 931
rand функция, возврат, 831
хранение в компьютере, порядок, 680
подчеркивание в, 95
поразрядные операторы, действие, 139
преобразование нечисловых переменных в, 400
случайные, 831, 863
сравнения операторы, 60
строки, преобразование в, 94

числа
типизация в Perl, 94
усечение с помощью int, 792
функции для, 739
целые, 1082
цифры в именах, 90
(цифра) метасимвол шаблона, 70
шестнадцатеричные, 95, 195, 791
ширина, 680
числовые литералы, 95
числовой контекст, 106, 1060
чтение
данных через системный вызов низкого уровня, 874
сегмента совместно используемой памяти, 849
файлов, режим открытия и, 811

Ш

шаблоны (и регулярные выражения), 68, 179, 801
grep функция и, 791
re прагма, изменение режима по умолчанию, 936
split функция и, 185, 857
study функция, 865
авторасщепление с помощью ключа -a, 546
алфавитно-цифровые метазнаки, 203
выполнение кода на этапе поиска, 255
вычисление при подстановке, 254
генерируемые, 252
декларативный подход, 252
жадный поиск, 71, 244
завершающий ограничитель для, 184
захват и кластеризация, 225
изменения между версиями Perl, 646
интерполяция шаблона на этапе поиска, 259
использование кавычек, 98
квантификаторы, 182, 201, 219, 244
классы символов, 70, 208
путаница с индексами массивов, 101
кластеризация в, 228
код Perl внутри, 252
шаблоны
метасимволы и метазнаки, 199, 206
минимальное соответствие, 72
модификаторы, 188
замкнутые, 229
отладчик, команды поиска, 565
позиции для поиска, 222
поиск
$`, $&, $' переменные и, 187, 193
операторы для, 68, 183
повторяющихся слов в абзацах, 190
подстроки, соответствующие подшаблонам, 187
приоритетность, 241
рекурсивный, 259
функции для, 739
свойства Unicode, поиск, 457
символы в, 180
символы, соответствие вместо байтов, 457
синтаксис расширений, 202
заглядывающие утверждения, 247
сложные шаблоны, 247
снятия мечености данных для, 614
соображения безопасности, 636
ссылки на найденный текст, 74, 226
стиль программирования, 251
строки, 819
символы для pack/unpack, 820
управление поиском, 232
компилятор регулярных выражений, 239
последовательность выполнения команд Perl, 232
условная интерполяция в, 260
утверждения в, 181
определение собственных, 261
шестнадцатеричные числа, 95, 791, 1082
номер символа, задание, 206
преобразование чисел в, 195
ширина
представление символов, 453
чисел, 680
широковещание, 1082
шифрование, 755
модули Perl для, 936
шрифты, языки, проверка для, 214

Э

экземпляры, 355, 1082
экспоненциальная нотация, 95
экспорт
имен, 346, 349, 350
без использования метода import из Exporter, 351
символов, 1082
элементы массивов, 40, 1082
этап компиляции, 1083
сравнение с фазой компиляции, 518
управление отладчиком, 560
эффективность и программы Perl, 648, 657

Я

явные вызовы методов, 359
ядро
Unix, ошибка сценариев set-id, 623
автоматическая уборка зомби, 466
генерация сигналов, 463
надежность сигналов в, 467
потоки и, 500
языки
естественные и искусственные, 34
проверка символов для, 214
свободной формы, 83
трансляция, 675

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 5-93286-020-0, название «Программирование на Perl, 3-е издание» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.