HIGH TECH

ПОЛ ГРЭМ

ANSI Common LISP

СИМВОЛ®

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 978-5-93286-206-3, название «ANSI Common Lisp» – 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.

ANSI Common Lisp

Paul Graham

PRENTICE HALL

HIGH TECH

ANSI Common Lisp

Пол Грэм

СИМВОЛ®

Санкт-Петербург — Москва
2012

Серия «High tech»
Пол Грэм
ANSI Common Lisp
Перевод И. Хохлов

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*В. Демкин*
Редактор	*А. Родин*
Корректор	*С. Беляева*
Верстка	*Д. Орлова*

Грэм П.

ANSI Common Lisp. – Пер. с англ. – СПб.: Символ-Плюс, 2012. – 448 с., ил.

ISBN 978-5-93286-206-3

Книга «ANSI Common Lisp» сочетает в себе введение в программирование на Лиспе и актуальный справочный материал по ANSI-стандарту языка. Новички найдут в ней примеры интересных программ с их тщательным объяснением. Профессиональные разработчики оценят всесторонний практический подход.

Автор постарался показать уникальные особенности, которые выделяют Лисп из множества других языков программирования, а также предоставляемые им новые возможности, например макросы, которые позволяют разработчику писать программы, которые будут писать другие программы. Лисп – единственный язык, который позволяет с легкостью осуществлять это, потому что только он предлагает необходимые для этого абстракции.

Книга содержит: детальное рассмотрение объектно-ориентированного программирования – не только описание CLOS, но и пример собственного встроенного объектно-ориентированного языка; более 20 самостоятельных примеров, в том числе трассировщик лучей, генератор случайного текста, сопоставление с образцом, логический вывод, программа для генерации HTML, алгоритмы поиска и сортировки, файлового ввода-вывода, сжатия данных, а также вычислительные задачи. Особое внимание уделяется критически важным концепциям, включая префиксный синтаксис, связь кода и данных, рекурсию, функциональное программирование, типизацию, неявное использование указателей, динамическое выделение памяти, замыкания, макросы, предшествование классов, суть методов обобщенных функций и передачи сообщений. Вы найдете полноценное руководство по оптимизации, примеры различных стилей программирования, включая быстрое прототипирование, разработку снизу-вверх, объектно-ориентированное программирование и применение встраиваемых языков.

ISBN 978-5-93286-206-3
ISBN 0-13-370875-6 (англ)

© Издательство Символ-Плюс, 2012
Authorized translation of the English edition © 1996 Prentice Hall, Inc. This translation is published and sold by permission of Prentice Hall, 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». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 380-5007, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Подписано в печать 03.10.2012. Формат 70×100 1/16.
Печать офсетная. Объем 28 печ. л.

Оглавление

Предисловие . . . 13

Предисловие к русскому изданию . . . 17

1. Введение . . . 19

1.1. Новые инструменты . . . 19
1.2. Новые приемы . . . 21
1.3. Новый подход . . . 22

2. Добро пожаловать в Лисп . . . 25

2.1. Форма . . . 25
2.2. Вычисление . . . 27
2.3. Данные . . . 28
2.4. Операции со списками . . . 30
2.5. Истинность . . . 30
2.6. Функции . . . 32
2.7. Рекурсия . . . 33
2.8. Чтение Лиспа . . . 34
2.9. Ввод и вывод . . . 35
2.10. Переменные . . . 37
2.11. Присваивание . . . 38
2.12. Функциональное программирование . . . 39
2.13. Итерация . . . 40
2.14. Функции как объекты . . . 42
2.15. Типы . . . 44
2.16. Заглядывая вперед . . . 44
Итоги главы . . . 45
Упражнения . . . 46

3. Списки . . . 48

3.1. Ячейки . . . 48
3.2. Равенство . . . 50
3.3. Почему в Лиспе нет указателей . . . 51
3.4. Построение списков . . . 53
3.5. Пример: сжатие . . . 53
3.6. Доступ . . . 55
3.7. Отображающие функции . . . 56

3.8. Деревья . . . 57
3.9. Чтобы понять рекурсию, нужно понять рекурсию . . . 58
3.10. Множества . . . 60
3.11. Последовательности . . . 61
3.12. Стопка . . . 63
3.13. Точечные пары . . . 65
3.14. Ассоциативные списки . . . 66
3.15. Пример: поиск кратчайшего пути . . . 67
3.16. Мусор . . . 69
Итоги главы . . . 70
Упражнения . . . 71

4. Специализированные структуры данных . . . 73

4.1. Массивы . . . 73
4.2. Пример: бинарный поиск . . . 75
4.3. Строки и знаки . . . 77
4.4. Последовательности . . . 78
4.5. Пример: разбор дат . . . 81
4.6. Структуры . . . 83
4.7. Пример: двоичные деревья поиска . . . 85
4.8. Хеш-таблицы . . . 90
Итоги главы . . . 93
Упражнения . . . 94

5. Управление . . . 95

5.1. Блоки . . . 95
5.2. Контекст . . . 97
5.3. Условные выражения . . . 99
5.4. Итерации . . . 100
5.5. Множественные значения . . . 103
5.6. Прерывание выполнения . . . 104
5.7. Пример: арифметика над датами . . . 106
Итоги главы . . . 109
Упражнения . . . 110

6. Функции . . . 111

6.1. Глобальные функции . . . 111
6.2. Локальные функции . . . 112
6.3. Списки параметров . . . 113
6.4. Пример: утилиты . . . 115
6.5. Замыкания . . . 118
6.6. Пример: строители функций . . . 120
6.7. Динамический диапазон . . . 123
6.8. Компиляция . . . 124
6.9. Использование рекурсии . . . 125
Итоги главы . . . 128
Упражнения . . . 128

7. Ввод и вывод 130
7.1. Потоки 130
7.2. Ввод 132
7.3. Вывод 134
7.4. Пример: замена строк 136
7.5. Макрознаки 141
Итоги главы 142
Упражнения 142

8. Символы 144
8.1. Имена символов 144
8.2. Списки свойств 145
8.3. А символы-то не маленькие 146
8.4. Создание символов 146
8.5. Использование нескольких пакетов 147
8.6. Ключевые слова 148
8.7. Символы и переменные 149
8.8. Пример: генерация случайного текста 149
Итоги главы 152
Упражнения 152

9. Числа 154
9.1. Типы 154
9.2. Преобразование и извлечение 155
9.3. Сравнение 157
9.4. Арифметика 158
9.5. Возведение в степень 159
9.6. Тригонометрические функции 159
9.7. Представление 160
9.8. Пример: трассировка лучей 161
Итоги главы 168
Упражнения 169

10. Макросы 170
10.1. Eval 170
10.2. Макросы 172
10.3. Обратная кавычка 173
10.4. Пример: быстрая сортировка 174
10.5. Проектирование макросов 175
10.6. Обобщенные ссылки 178
10.7. Пример: макросы-утилиты 179
10.8. На Лиспе 182
Итоги главы 183
Упражнения 183

11. CLOS 185

11.1. Объектно-ориентированное программирование 185
11.2. Классы и экземпляры 187
11.3. Свойства слотов 188
11.4. Суперклассы 190
11.5. Предшествование 190
11.6. Обобщенные функции 192
11.7. Вспомогательные методы 195
11.8. Комбинация методов 197
11.9. Инкапсуляция 198
11.10. Две модели 199
Итоги главы 200
Упражнения 201

12. Структура 202

12.1. Разделяемая структура 202
12.2. Модификация 205
12.3. Пример: очереди 206
12.4. Деструктивные функции 208
12.5. Пример: двоичные деревья поиска 209
12.6. Пример: двусвязные списки 212
12.7. Циклическая структура 215
12.8. Неизменяемая структура 217
Итоги главы 218
Упражнения 218

13. Скорость 220

13.1. Правило бутылочного горлышка 220
13.2. Компиляция 221
13.3. Декларации типов 224
13.4. Обходимся без мусора 229
13.5. Пример: заранее выделенные наборы 232
13.6. Быстрые операторы 234
13.7. Две фазы разработки 236
Итоги главы 237
Упражнения 238

14. Более сложные вопросы 239

14.1. Спецификаторы типов 239
14.2. Бинарные потоки 241
14.3. Макросы чтения 241
14.4. Пакеты 243
14.5. Loop 246
14.6. Особые условия 250

15. Пример: логический вывод 253
15.1. Цель 253
15.2. Сопоставление 254
15.3. Отвечая на запросы 256
15.4. Анализ 261

16. Пример: генерация HTML 263
16.1. HTML 263
16.2. Утилиты HTML 265
16.3. Утилита для итерации 268
16.4. Генерация страниц 269

17. Пример: объекты 274
17.1. Наследование 274
17.2. Множественное наследование 276
17.3. Определение объектов 278
17.4. Функциональный синтаксис 279
17.5. Определение методов 280
17.6. Экземпляры 281
17.7. Новая реализация 282
17.8. Анализ 288

A. Отладка 290

B. Лисп на Лиспе 299

C. Изменения в Common Lisp 307

D. Справочник по языку 314

Комментарии 421

Алфавитный указатель 436

Half lost on my firmness gains to more glad heart,
Or violent and from forage drives
A glimmering of all sun new begun
Both harp thy discourse they march'd,
Forth my early, is not without delay;
For their soft with whirlwind; and balm.
Undoubtedly he scornful turn'd round ninefold,
Though doubled now what redounds,
And chains these a lower world devote, yet inflicted?
Till body or rare, and best things else enjoy'd in heav'n
To stand divided light at ev'n and poise their eyes,
Or nourish, lik'ning spiritual, I have thou appear. [1]

Henley

Я потерял итог моих трудов, что сердце грели мне,
И было то ожесточенье сердца иль движение вперед,
Но солнце снова озарило нас,
И арфа вновь твоя заговорила,
Вперед, как можно раньше и без промедленья;
Тот ураган для них стал мягок, как бальзам.
Он тень сомнения отверг и обвился вокруг с насмешкой девять раз,
Наперекор тому, что дважды отразилось эхом,
И цепи эти уходили к преисподней, разве не ужасно?
Покуда тело дивное испытывало райские услады
Стоять мне, рассеченным светом, и взглядом воспарить,
Иль вскормленный, подобно духам, я появился пред тобою.

Henley

---

[1] Эта строфа написана программой Henley на основе поэмы Джона Мильтона «Потерянный рай». Можете попытаться найти тут смысл, но лучше загляните в главу 8. – *Прим. перев.*

Предисловие

Цель данной книги – быстро и основательно научить вас языку Common Lisp. Книга состоит из двух частей: в первой части на множестве примеров объясняются основные концепции программирования на Common Lisp, вторая часть – это современное описание стандарта ANSI Common Lisp, содержащее каждый оператор языка.

Аудитория

Книга «ANSI Common Lisp» предназначена как для студентов, изучающих этот язык, так и для профессиональных программистов. Ее чтение не требует предварительного знания Лиспа. Опыт написания программ на других языках был бы, безусловно, полезен, но не обязателен. Повествование начинается с основных понятий, что позволяет уделить особое внимание моментам, которые обычно приводят в замешательство человека, впервые знакомящегося с Лиспом.

Эта книга может использоваться в качестве учебного пособия по Лиспу или как часть курсов, посвященных искусственному интеллекту или теории языков программирования. Профессиональные разработчики, желающие изучить Лисп, оценят простой, практический подход. Те, кто уже знаком с языком, найдут в книге множество полезных примеров и оценят ее как удобный справочник по стандарту ANSI Common Lisp.

Как пользоваться книгой

Лучший способ выучить Лисп – использовать его. Кроме того, намного интереснее изучать язык в процессе написания программ. Книга устроена так, чтобы читатель смог начать делать это как можно раньше. После небольшого введения в главе 2 объясняется все, что понадобится для создания первых Лисп-программ. В главах 3–9 рассматриваются ключевые элементы программирования на Лиспе. Особое внимание уделяется таким понятиям, как роль указателей в Лиспе, использование рекурсии и значимость функций как полноценных объектов языка.

Следующие материалы предназначены для читателей, которые хотят более основательно разобраться с техникой программирования на Лиспе. Главы 10–14 охватывают макросы, CLOS (объектная система Common

Lisp), операции со списками, оптимизацию, а также более сложные темы, такие как пакеты и макросы чтения. Главы 15–17 подводят итог предыдущих глав на трех примерах реальных приложений: программа для создания логических интерфейсов, HTML-генератор и встроенный объектно-ориентированный язык программирования.

Последняя часть книги состоит из четырех приложений, которые будут полезны всем читателям. Приложения A–D включают руководство по отладке, исходные коды для 58 операторов языка, описание основных отличий ANSI Common Lisp от предыдущих версий языка° и справочник по каждому оператору в ANSI Common Lisp.

Книга завершается комментариями, содержащими пояснения, ссылки, дополнительный код и прочие отступления. Наличие комментария помечается в основном тексте маленьким кружочком: °.

Код

Несмотря на то что книга посвящена ANSI Common Lisp, по ней можно изучать любую разновидность Common Lisp. Примеры, демонстрирующие новые возможности, обычно сопровождаются комментариями, поясняющими, как они могут быть адаптированы к более ранним реализациям.

Весь код из книги, ссылки на свободный софт, исторические документы, часто задаваемые вопросы и множество других ресурсов доступны по адресу:

http://www.eecs.harvard.edu/onlisp/

Анонимный доступ к коду можно получить по ftp:

ftp://ftp.eecs.harvard.edu:/pub/onlisp/

Вопросы и комментарии присылайте на *pg@eecs.harvard.edu*.

«On Lisp»

В этой книге я постарался показать уникальные особенности, которые выделяют Лисп из множества языков программирования, а также предоставляемые им новые возможности. Например, макросы – они позволяют разработчику писать программы, которые будут писать другие программы. Лисп – единственный язык, который позволяет с легкостью осуществлять это, потому что только он предлагает необходимые для этого абстракции. Читателям, которым интересно узнать больше о макросах и других интересных возможностях языка, я предлагаю познакомиться с книгой «On Lisp»[1], которая является продолжением данного издания.

[1] Paul Graham «On Lisp», Prentice Hall, 1993. – *Прим. перев.*

Благодарности

Из всех друзей, участвовавших в работе над книгой, наибольшую благодарность я выражаю Роберту Моррису, который на протяжении всего времени помогал совершенствовать это издание. Некоторые примеры, включая Henley (стр. 149) и сопоставление с образцом (стр. 254), взяты из написанного им кода.

Я был счастлив работать с первоклассной командой технических рецензентов: Сконом Бриттаном (Skona Brittain), Джоном Фодераро (John Foderaro), Ником Левином (Nick Levine), Питером Норвигом (Peter Norvig) и Дэйвом Турецки (Dave Touretzky). Вряд ли найдется хотя бы одна страница, к улучшению которой они не приложили руку. Джон Фодераро даже переписал часть кода к разделу 5.7.

Нашлись люди, которые согласились прочитать рукопись целиком или частично, включая Кена Андерсона (Ken Anderson), Тома Читама (Tom Cheatham), Ричарда Фейтмана (Richard Fateman), Стива Хайна (Steve Hain), Барри Марголина (Barry Margolin), Уолдо Пачеко (Waldo Pacheco), Вилера Румла (Wheeler Ruml) и Стюарта Рассела (Stuart Russel). Кен Андерсон и Вилер Румл сделали множество полезных замечаний.

Я благодарен профессору Читаму и Гарварду в целом за предоставление необходимых для написания книги возможностей. Также благодарю сотрудников лаборатории Айкен: Тони Хэртмана (Tony Hartman), Дейва Мазиерса (Dave Mazieres), Януша Джуду (Janusz Juda), Гарри Бохнера (Harry Bochner) и Джоанну Клис (Joanna Klys).

Я рад, что у меня снова появилась возможность поработать с сотрудниками Prentice Hall: Аланом Аптом (Alan Apt), Моной Помпили (Mona Pompili), Ширли МакГир (Shirley McGuire) и Ширли Майклс (Shirley Michaels). Обложка книги выполнена превосходным мастером Джино Ли (Gino Lee) из Bow & Arrow Press, Кэмбридж.

Эта книга была набрана с помощью LaTeX, языка, созданного Лесли Лэмпортом (Leslie Lamport) поверх TeX'а Дональда Кнута (Donald Knuth), с использованием дополнительных макросов авторства Л.А. Карра (L.A. Carr), Вана Якобсона (Van Jacobson) и Гая Стила (Guy Steele). Чертежи были выполнены с помощью программы Idraw, созданной Джоном Влиссидсом (John Vlissids) и Скоттом Стантоном (Skott Stanton). Предпросмотр всей книги был сделан с помощью программы Ghostview, написанной Тимом Тейзеном (Tim Theisen) на основе интерпретатора Ghostscript, созданного Л. Питером Дойчем (L. Peter Deutch).

Я должен поблагодарить и многих других людей: Генри Бейкера (Henry Baker), Кима Барретта (Kim Barrett), Ингрид Бассет (Ingrid Basset), Тревора Блеквелла (Trevor Blackwell), Пола Беккера (Paul Becker), Гарри Бисби (Gary Bisbee), Франка Душмана (Frank Dueschmann), Франсиса Дикли (Frances Dickey), Рича и Скотта Дрейвса (Rich and Scott Draves), Билла Дабак (Bill Dubuque), Дэна Фридмана (Dan Friedman),

Дженни Грэм (Jenny Graham), Эллис Хартли (Alice Hartley), Дэвида Хендлера (David Hendler), Майка Хьюлетта (Mike Hewlett), Гленна Холловота (Glenn Hollowat), Брэда Карпа (Brad Karp), Соню Кини (Sonya Keene), Росса Найтс (Ross Knights), Митсуми Комуро (Mutsumi Komuro), Стефи Кутзиа (Steffi Kutzia), Дэвида Кузника (David Kuznick), Мэди Лорд (Madi Lord), Джулию Маллози (Julie Mallozzi), Пола МакНами (Paul McNamee), Дэйва Муна (Dave Moon), Говарда Миллингса (Howard Mullings), Марка Ницберга (Mark Nitzberg), Ненси Пармет (Nancy Parmet) и ее семью, Роберта Пенни (Robert Penny), Майка Плуча (Mike Plusch), Шерил Сэкс (Cheryl Sacks), Хейзэма Сейеда (Hazem Sayed), Шеннона Спайреса (Shannon Spires), Лоу Штейнберга (Lou Steinberg), Пола Стоддарда (Paul Stoddard), Джона Стоуна (John Stone), Гая Стила (Guy Steele), Стива Страссмана (Steve Strassmann), Джима Вейча (Jim Veitch), Дэйва Уоткинса (Dave Watkins), Айдела и Джулианну Вебер (Idelle and Julian Weber), Вейкерсов (the Weickers), Дэйва Йоста (Dave Yost) и Алана Юлли (Alan Yuille).

Но больше всего я благодарен моим родителям и Джекки.

Дональд Кнут назвал свою известную серию книг «Искусство программирования». В прочитанной им Тьюринговской лекции он объяснил, что это название было выбрано не случайно, так как в программирование его привела как раз «возможность писать красивые программы».

Так же как и архитектура, программирование сочетает в себе искусство и науку. Программа создается на основе математических принципов, так же как здание держится согласно законам физики. Но задача архитектора не просто построить здание, которое не разрушится. Почти всегда он стремится создать нечто прекрасное.

Как и Дональд Кнут, многие программисты чувствуют, что такова истинная цель программирования. Так считают почти все Лисп-хакеры. Саму суть лисп-хакерства можно выразить двумя фразами. Программирование должно доставлять радость. Программы должны быть красивыми. Таковы идеи, которые я постарался пронести через всю книгу.

Пол Грэм

Предисловие к русскому изданию

Книга, перевод которой вы держите в руках, была издана в 1996 году, а написана и того раньше. К моменту подготовки перевода прошло 15 лет. Для такой стремительно развивающейся отрасли, как программирование, это огромный срок, за который изменили облик не только парадигмы и языки программирования, но и сама вычислительная техника.

Несмотря на это, данная книга и на настоящий момент представляет большую практическую ценность. Она соответствует стандарту языка, который не менялся с момента ее написания и, похоже, не будет меняться в течение ощутимого времени. Кроме того, в книге описаны модели и методы, пришедшие в программирование из Лиспа и в той или иной мере актуальные в современном программировании.

Автор не раз упоминает о том, что Лисп, несмотря на его долгую историю, не теряет актуальности. Теперь, когда с момента издания оригинала книги прошло 15 лет, а с момента создания языка Лисп более полувека, мы отчетливо видим подтверждение слов автора, наблюдая постоянный рост интереса к языку.

Тем не менее некоторые моменты в книге являются слегка устаревшими и требуют дополнительных комментариев.

В числе уникальных особенностей Лиспа Грэм выделяет интерактивность, автоматическое управление памятью, динамическую типизацию и замыкания. На момент написания книги Лисп конкурировал с такими языками, как C, C++, Паскаль, Фортран (на протяжении книги автор сравнивает Лисп именно с ними). Эти языки «старой закалки» действительно представляют полную противоположность Лиспу. На настоящий момент разработано множество языков, в которых в той или иной степени заимствованы преимущества Лиспа. Таким, например, является Perl, который вытесняется более продвинутым языком Python, а последний, несмотря на популярность, сам испытывает конкуренцию со стороны языка Ruby, известного как «Лисп с человеческим синтаксисом». Такие языки благодаря гибкости быстро находят свою нишу, оставаясь при этом средствами общего назначения. Так, Perl прочно занял нишу скриптового языка в Unix-подобных системах. Однако механизм макросов, лежащий в основе Лиспа, пока не был заимствован ни одним из языков, так как прочно связан с его синтаксисом. Кроме того, Лисп выгодно отличается от своих «последователей». Согласитесь, искусственное

добавление возможностей в язык с уже существующей структурой и идеологией существенно отличается от случая, когда язык изначально разрабатывался с учетом данных возможностей.

Время коснулось также и ряда идей и моделей, упомянутых в данной книге. Несколько странными могут показаться восторженные упоминания об объектно-ориентированном программировании. Прошло немало времени, и сегодня ООП уже больше не вызывает подобный восторг.

Многое изменилось и в мире реализаций Common Lisp. Автор сознательно не упоминает названия реализаций, так как их жизненный срок не определен. Многих реализаций языка уже нет в живых, но на их место пришли новые. Необходимо отметить, что сейчас имеется ряд блестящих реализаций Common Lisp, как коммерческих, так и свободных. Стандарт языка дает разработчикам довольно много свободы действий, и выпускаемые ими реализации как внешне, так и внутренне могут сильно отличаться друг от друга. Детали реализаций вас могут не волновать, а вот различия в их поведении могут смущать новичков. Когда речь заходит о взаимодействии с пользователем (например, о работе в отладчике), автор использует некий упрощенный унифицированный интерфейс, который он называет «гипотетической» реализацией. На деле, вам придется поэкспериментировать с выбранной реализацией, чтобы научиться эффективно ее использовать. Кроме того, сейчас имеется отличная среда разработки Slime[1], помимо прочего скрывающая разницу в поведении между реализациями.

Всеволод Демкин

Переводчик, Иван Хохлов, выражает благодарность Ивану Струкову, Сергею Катревичу и Ивану Чернецкому за предоставление ценных замечаний по переводу отдельных глав книги.

[1] Домашняя страница проекта – *http://common-lisp.net/project/slime/*.

1

Введение

Джон Маккарти со своими студентами начал работу над первой реализацией Лиспа в 1958 году. Не считая Фортрана, Лисп – это старейший из ныне используемых языков.° Но намного важнее то, что до сих пор он остается флагманом среди языков программирования. Специалисты, хорошо знающие Лисп, скажут вам, что в нем есть нечто, делающее его особенным по сравнению с остальными языками.

Отчасти его отличает изначально заложенная возможность развиваться. Лисп позволяет программисту определять новые операторы, и если появятся новые абстракции, которые приобретут популярность (например, объектно-ориентированное программирование), их всегда можно будет реализовать в Лиспе. Изменяясь как ДНК, такой язык никогда не выйдет из моды.

1.1. Новые инструменты

Зачем изучать Лисп? Потому что он позволяет делать то, чего не могут другие языки. Если вы захотите написать функцию, складывающую все числа, меньшие *n*, она будет очень похожа на аналогичную функцию в C:

```
; Lisp                    /* C */
(defun sum (n)            int sum(int n){
  (let ((s 0))              int i , s = 0;
    (dotimes (i n s)        for(i = 0; i < n; i++)
      (incf s i))))           s += i;
                            return(s);
                          }
```

Если вы хотите делать несложные вещи типа этой, в сущности, не имеет значения, какой язык использовать. Предположим, что теперь мы

хотим написать функцию, которая принимает число *n* и возвращает функцию, которая добавляет *n* к своему аргументу:

```
; Lisp
(defun addn (n)
  #'(lambda (x)
      (+ x n)))
```

Как функция addn будет выглядеть на C? Ее просто невозможно написать.

Вы, вероятно, спросите, зачем это может понадобиться? Языки программирования учат вас не хотеть того, что они не могут осуществить. Раз программисту приходится думать на том языке, который он использует, ему сложно представить то, чего он не может описать. Когда я впервые занялся программированием на Бейсике, я не огорчался отсутствием рекурсии, так как попросту не знал, что такая вещь имеет место. Я думал на Бейсике и мог представить себе только итеративные алгоритмы, так с чего бы мне было вообще задумываться о рекурсии?

Если вам не нужны лексические замыкания (а именно они были продемонстрированы в предыдущем примере), просто примите пока что на веру, что Лисп-программисты используют их постоянно. Сложно найти программу на Лиспе, написанную без использования замыканий. В разделе 6.7 вы научитесь пользоваться ими.

Замыкания – не единственные абстракции, которых нет в других языках. Другой, возможно даже более важной, особенностью Лиспа является то, что программы, написанные на нем, представляются в виде его же структур данных. Это означает, что вы можете писать программы, которые пишут программы. Действительно ли люди пользуются этим? Да, это и есть макросы, и опытные программисты используют их на каждом шагу. Вы узнаете, как создавать свои макросы, в главе 10.

С макросами, замыканиями и динамической типизацией Лисп превосходит объектно-ориентированное программирование. Если вы в полной мере поняли предыдущее предложение, то, вероятно, можете не читать эту книгу. Это важный момент, и вы найдете подтверждение тому в коде к главе 17.

Главы 2–13 поэтапно вводят все понятия, необходимые для понимания кода главы 17. Благодаря вашим стараниям вы будете ощущать программирование на C++ таким же удушающим, каким опытный программист C++ в свою очередь ощущает Бейсик. Сомнительная, на первый взгляд, награда. Но, быть может, вас вдохновит осознание природы этого дискомфорта. Бейсик неудобен по сравнению с C++, потому что опытный программист C++ знает приемы, которые невозможно осуществить в Бейсике. Точно так же изучение Лиспа даст вам больше, нежели добавление еще одного языка в копилку ваших знаний. Вы научитесь размышлять о программах по-новому, более эффективно.

1.2. Новые приемы

Итак, Лисп предоставляет такие инструменты, которых нет в других языках. Но это еще не все. Отдельные технологии, впервые появившиеся в Лиспе, такие как автоматическое управление памятью, динамическая типизация, замыкания и другие, значительно упрощают программирование. Взятые вместе, они создают критическую массу, которая рождает новый подход к программированию.

Лисп изначально гибок – он позволяет самостоятельно задавать новые операторы. Это возможно сделать, потому что сам Лисп состоит из таких же функций и макросов, как и ваши собственные программы. Поэтому расширить возможности Лиспа ничуть не сложнее, чем написать свою программу. На деле это так просто (и полезно), что расширение языка стало обычной практикой. Получается, что вы не только пишете программу в соответствии с языком, но и дополняете язык в соответствии с нуждами программы. Этот подход называется *снизу-вверх* (*bottom-up*).

Практически любая программа будет выигрывать, если используемый язык заточен под нее, и чем сложнее программа, тем большую значимость имеет подход «снизу-вверх». В такой программе может быть несколько слоев, каждый из которых служит чем-то вроде языка для описания вышележащего слоя. Одной из первых программ, написанных таким образом, был TEX. Вы имеете возможность писать программы снизу-вверх на любом языке, но на Лиспе это делать проще всего.

Написанные снизу-вверх программы легко расширяемы. Поскольку идея расширяемости лежит в основе Лиспа, это идеальный язык для написания расширяемых программ. В качестве примера приведу три программы, написанные в 80-х годах и использовавшие возможность расширения Лиспа: GNU Emacs, Autocad и Interleaf.

Кроме того, код, написанный данным методом, легко использовать многократно. Суть написания повторно используемого кода заключается в отделении общего от частного, а эта идея лежит в самой основе метода. Вместо того чтобы прилагать существенные усилия к созданию монолитного приложения, полезно потратить часть времени на построение своего языка, поверх которого затем реализовывать само приложение. Приложение будет находиться на вершине пирамиды языковых слоев и будет иметь наиболее специфичное применение, а сами слои можно будет приспособить к повторному использованию. Действительно, что может быть более пригодным для многократного применения, чем язык программирования?

Лисп позволяет не просто создавать более утонченные приложения, но и делать это быстрее. Практика показывает, что программы на Лиспе выглядят короче, чем аналоги на других языках. Как показал Фредерик Брукс, временные затраты на написание программы зависят в первую очередь от ее длины.° В случае Лиспа этот эффект усиливается его

динамическим характером, за счет которого сокращается время между редактированием, компиляцией и тестированием.

Мощные абстракции и интерактивность вносят коррективы и в принцип разработки приложений. Суть Лиспа можно выразить одной фразой – *быстрое прототипирование*. На Лиспе гораздо удобнее и быстрее написать прототип, чем составлять спецификацию. Более того, прототип служит проверкой предположения, является ли эффективным выбранный метод, а также гораздо ближе к готовой программе, чем спецификация.

Если вы еще не знаете Лисп достаточно хорошо, то это введение может показаться набором громких и, возможно, бессмысленных утверждений. Превзойти объектно-ориентированное программирование? Подстраивать язык под свои программы? Программировать на Лиспе в реальном времени? Что означают все эти утверждения? До тех пор пока вы не познакомитесь с Лиспом поближе, все эти слова будут звучать для вас несколько противоестественно, однако с опытом придет и понимание.

1.3. Новый подход

Одна из целей этой книги – не просто объяснить Лисп, но и продемонстрировать новый подход к программированию, который стал возможен благодаря этому языку. Подход, который вы чаще будете видеть в будущем. С ростом мощности сред разработки и увеличением абстрактности языков стиль программирования на Лиспе постепенно заменяет старую модель, в которой реализации предшествовало проектирование.

Согласно этой модели баги вообще не должны появляться. Программа, созданная по старательно разработанным заранее спецификациям, работает отлично. В теории звучит неплохо. К сожалению, эти спецификации разрабатываются и реализуются людьми, а люди не застрахованы от ошибок и каких-либо упущений. Кроме того, тяжело учесть все нюансы на стадии проектирования. В результате такой метод часто не срабатывает.

Руководитель проекта OS/360 Фредерик Брукс был хорошо знаком с традиционным подходом, а также с его результатами:

> Любой пользователь OS/360 вскоре начинал понимать, насколько лучше могла бы быть система. Более того, продукт не успевал за прогрессом, использовал памяти больше запланированного, стоимость его в несколько раз превосходила ожидаемую, и он не работал стабильно до тех пор, пока не было выпущено несколько релизов.°

Так он описывал систему, ставшую тогда одной из наиболее успешных.

Проблема в том, что такой подход не учитывает человеческий фактор. При использовании старой модели вы ожидаете, что спецификации не содержат серьезных огрехов и что остается лишь каким-то образом оттранслировать их в код. Опыт показывает, что это ожидание редко бы-

вает оправдано. Гораздо полезнее предполагать, что спецификация будет реализована с ошибками, а в коде будет полно багов.

Это как раз та идея, на которой построен новый подход. Вместо того чтобы надеяться на безукоризненную работу программистов, она пытается минимизировать стоимость допущенных ошибок. Стоимость ошибки – это время, необходимое на ее исправление. Благодаря мощным языкам и эффективным инструментам это время может быть существенно уменьшено. Кроме того, разработчик больше не удерживается в рамках спецификации, меньше зависит от планирования и может экспериментировать.

Планирование – это необходимое зло. Это ответ на риск: чем он больше, тем важнее планировать наперед. Мощные инструменты уменьшают риск, в результате уменьшается и необходимость планирования. Дизайн вашей программы может быть улучшен на основании информации из, возможно, самого ценного источника – опыта ее реализации.

Лисп развивался в этом направлении с 1960 года. На Лиспе вы можете писать прототипы настолько быстро, что успеете пройти несколько итераций проектирования и реализации раньше, чем закончили бы составлять спецификацию в старой модели. Все тонкости реализации будут осознаны в процессе создания программы, поэтому переживания о багах пока можно отложить в сторону. В силу функционального подхода к программированию многие баги имеют локальный характер. Некоторые баги (переполнения буфера, висящие указатели) просто невозможны, а остальные проще найти, потому что программа становится короче. И когда у вас есть интерактивная разработка, можно исправить их мгновенно, вместо того чтобы проходить через длинный цикл редактирования, компиляции и тестирования.

Такой подход появился не просто так, он действительно приносит результат. Как бы странно это ни звучало, чем меньше планировать разработку, тем стройнее получится программа. Забавно, но такая тенденция наблюдается не только в программировании. В средние века, до изобретения масляных красок, художники пользовались особым материалом – темперой, которая не могла быть перекрашена или осветлена. Стоимость ошибки была настолько велика, что художники боялись экспериментировать. Изобретение масляных красок породило множество течений и стилей в живописи. Масло «позволяет думать дважды».° Это дало решающее преимущество в работе со сложными сюжетами, такими как человеческая натура.

Введение в обиход масляных красок не просто облегчило жизнь художникам. Стали доступными новые, прогрессивные идеи. Янсон писал:

> Без масла покорение фламандскими мастерами визуальной реальности было бы крайне затруднительным. С технической точки зрения они являются прародителями современной живописи, потому что масло стало базовым инструментом художника с тех пор.°

Как материал темпера не менее красива, чем масло, но широта полета фантазии, которую обеспечивают масляные краски, является решающим фактором.

В программировании наблюдается похожая идея. Новая среда – это «объектно-ориентированный динамический язык программирования»; говоря одним словом, Лисп. Но это вовсе не означает, что через несколько лет все программисты перейдут на Лисп, ведь для перехода к масляным краскам тоже потребовалось существенное время. Кроме того, по разным соображениям темпера используется и по сей день, а порой ее комбинируют с маслом. Лисп в настоящее время используется в университетах, исследовательских лабораториях, некоторых компаниях, лидирующих в области софт-индустрии. А идеи, которые легли в основу Лиспа (например, интерактивность, сборка мусора, динамическая типизация), все больше и больше заимствуются в популярных языках.

Более мощные инструменты убирают риск из исследования. Это хорошая новость для программистов, поскольку она означает, что можно будет браться за более амбициозные проекты. Изобретение масляных красок, без сомнения, имело такой же эффект, поэтому период сразу после их внедрения стал золотым веком живописи. Уже есть признаки того, что подобное происходит и в программировании.

2

Добро пожаловать в Лисп

Цель этой главы – помочь вам начать программировать как можно скорее. Прочитав ее, вы узнаете достаточно о языке Common Lisp, чтобы начать писать программы.

2.1. Форма

Лисп – интерактивный язык, поэтому наилучший способ его изучения – в процессе использования. Любая Лисп-система имеет интерактивный интерфейс, так называемый *верхний уровень* (*toplevel*). Программист набирает выражения в *toplevel*, а система показывает их значения.

Чтобы сообщить, что система ожидает новые выражения, она выводит приглашение. Часто в качестве такого приглашения используется символ >. Мы тоже будем пользоваться им.

Одними из наиболее простых выражений в Лиспе являются целые числа. Если ввести 1 после приглашения,

```
> 1
1
>
```

система напечатает его значение, после чего выведет очередное приглашение, ожидая новых выражений.

В данном случае введенное выражение выглядит так же, как и полученное значение. Такие выражения называют самовычисляемыми. Числа (например, 1) – самовычисляемые объекты. Давайте посмотрим на более интересные выражения, вычисление которых требует совершения некоторых действий. Например, если мы хотим сложить два числа, то напишем

```
> (+ 2 3)
5
```

В выражении (+ 2 3) знак + – это *оператор*, а числа 2 и 3 – его *аргументы*.

В повседневной жизни вы бы написали это выражение как 2+3, но в Лиспе мы сначала ставим оператор +, следом за ним располагаем аргументы, а все выражение заключаем в скобки: (+ 2 3). Такую структуру принято называть *префиксной* нотацией, так как первым располагается оператор. Поначалу этот способ записи может показаться довольно странным, но на самом деле именно такому способу записи выражений Лисп обязан своими возможностями.

Например, если мы хотим сложить три числа, в обычной форме записи нам придется воспользоваться плюсом дважды:

```
2 + 3 + 4
```

в то время как в Лиспе вы всего лишь добавляете еще один аргумент:

```
(+ 2 3 4)
```

В обычной нотации[1] оператор + имеет два аргумента, один перед ним и один после. Префиксная запись дает большую гибкость, позволяя оператору иметь любое количество аргументов или вообще ни одного:

```
> (+)
0
> (+ 2)
2
> (+ 2 3)
5
> (+ 2 3 4)
9
> (+ 2 3 4 5)
14
```

Поскольку у оператора может быть произвольное число аргументов, нужен способ показать, где начинается и заканчивается выражение. Для этого используются скобки.

Выражения могут быть вложенными:

```
> (/ (- 7 1) (- 4 2))
3
```

Здесь мы делим разность 7 и 1 на разность 4 и 2.

Другая особенность префиксной нотации: все выражения в Лиспе – либо *атомы* (например, 1), либо *списки*, состоящие из произвольного количества выражений. Приведем допустимые Лисп-выражения:

```
2       (+ 2 3)       (+ 2 3 4)       (/ (- 7 1) (- 4 2))
```

[1] Ее также называют *инфиксной*. – *Прим. перев.*

В дальнейшем вы увидите, что весь код в Лиспе имеет такой вид. Языки типа C имеют более сложный синтаксис: арифметические выражения имеют инфиксную запись, вызовы функций записываются с помощью разновидности префиксной нотации, их аргументы разделяются запятыми, выражения отделяются друг от друга с помощью точки с запятой, а блоки кода выделяются фигурными скобками. В Лиспе же для выражения всех этих идей используется единая нотация.

2.2. Вычисление

В предыдущем разделе мы набирали выражения в *toplevel*, а Лисп выводил их значения. В этом разделе мы поближе познакомимся с процессом вычисления выражений.

В Лиспе `+` – это функция, а выражение вида `(+ 2 3)` – это вызов функции. Результат вызова функции вычисляется в 2 шага:

1. Сначала вычисляются аргументы, слева направо. В нашем примере все аргументы самовычисляемые, поэтому их значениями являются `2` и `3`.
2. Затем аргументы применяются к функции, задаваемой оператором. В нашем случае это оператор сложения, который возвращает `5`.

Аргумент может быть не только самовычисляемым объектом, но и другим вызовом функции. В этом случае он вычисляется по тем же правилам. Посмотрим, что происходит при вычислении выражения `(/ (- 7 1) (- 4 2))`:

1. Вычисляется `(- 7 1)`: `7` вычисляется в `7`, `1` – в `1`. Эти аргументы передаются функции `-`, которая возвращает `6`.
2. Вычисляется `(- 4 2)`: `4` вычисляется в `4`, `2` – в `2`, функция `-` применяется к этим аргументам и возвращает `2`.
3. Значения `6` и `2` передаются функции `/`, которая возвращает `3`.

Большинство операторов в Common Lisp – это функции, но не все. Вызовы функций всегда обрабатываются подобным образом. Аргументы вычисляются слева направо и затем передаются функции, которая возвращает значение всего выражения. Этот порядок называется *правилом вычисления* для Common Lisp.

Тем не менее существуют операторы, которые не следуют принятому в Common Lisp порядку вычислений. Один из них – `quote`, или оператор цитирования. `quote` – это *специальный оператор*; это означает, что у него есть собственное правило вычисления, а именно: ничего не делать. Фактически `quote` берет один аргумент и просто возвращает его текстовую запись:

```
> (quote (+ 3 5))
(+ 3 5)
```

Решение проблемы

Если вы ввели что-то, что Лисп не понимает, то он выведет сообщение об ошибке, и среда переведет вас в вариант toplevel, называемый *циклом прерывания* (*break loop*). Он дает опытному программисту возможность выяснить причину ошибки, но вам пока что потребуется знать только то, как выйти из этого цикла. В разных реализациях Common Lisp выход может осуществляться по-разному. В гипотетической реализации вам поможет команда `:abort`.

```
> (/ 1 0)
Error: Division by zero.
       Options: :abort, :backtrace
>> :abort
>
```

В приложении А показано, как отлаживать программы на Лиспе, а также приводятся примеры наиболее распространенных ошибок.

Для удобства в Common Lisp можно заменять оператор `quote` на кавычку. Тот же результат можно получить, просто поставив ' перед цитируемым выражением:

```
> '(+ 3 5)
(+ 3 5)
```

В явном виде оператор `quote` почти не используется, более распространена его сокращенная запись с кавычкой.

Цитирование в Лиспе является способом *защиты* выражения от вычисления. В следующем разделе будет показано, чем такая защита может быть полезна.

2.3. Данные

Лисп предоставляет все типы данных, которые есть в большинстве других языков, а также некоторые другие, отсутствующие где-либо еще. С одним типом данных мы уже познакомились. Это *integer* – целое число, записываемое в виде последовательности цифр: `256`. Другой тип данных, который есть в большинстве других языков, – *строка*, представляемая как последовательность символов, окруженная двойными кавычками: `"ora et labora"`. Так же как и целые числа, строки самовычисляемы.

В Лиспе есть два типа, которые редко используются в других языках, – символы и списки. *Символы* – это слова. Обычно они преобразуются к верхнему регистру независимо от того, как вы их ввели:

```
> 'Artichoke
ARTICHOKE
```

Символы, как правило, не являются самовычисляемым типом, поэтому, чтобы сослаться на символ, его необходимо цитировать, как показано выше.

Список – это последовательность из нуля или более элементов, заключенных в скобки. Эти элементы могут принадлежать к любому типу, в том числе могут являться другими списками. Чтобы Лисп не счел список вызовом функции, его нужно процитировать:

```
> '(my 3 "Sons")
(MY 3 "Sons")
> '(the list (a b c) has 3 elements)
(THE LIST (A B C) HAS 3 ELEMENTS)
```

Обратите внимание, что цитирование предотвращает вычисление всего выражения, включая все его элементы.

Список можно построить с помощью функции `list`. Как и у любой функции, ее аргументы вычисляются. В следующем примере внутри вызова функции `list` вычисляется значение функции +:

```
> (list 'my (+ 2 1) "Sons")
(MY 3 "Sons")
```

Пришло время оценить одну из наиболее важных особенностей Лиспа. *Программы, написанные на Лиспе, представляются в виде списков.* Если приведенные ранее доводы о гибкости и элегантности не убедили вас в ценности принятой в Лиспе нотации, то, возможно, этот момент заставит вас изменить свое мнение. Именно эта особенность позволяет программам, написанным на Лиспе, генерировать Лисп-код, что дает возможность разработчику создавать программы, которые пишут программы.

Хотя такие программы не рассматриваются в книге вплоть до главы 10, сейчас важно понять связь между выражениями и списками. Как раз для этого нам нужно цитирование. Если список цитируется, то результатом его вычисления будет сам список. В противном случае список будет расценен как код и будет вычислено его значение:

```
> (list '(+ 2 1) (+ 2 1))
((+ 2 1) 3)
```

Первый аргумент цитируется и остается списком, в то время как второй аргумент расценивается как вызов функции и превращается в число.

Список может быть пустым. В Common Lisp возможны два типа представления пустого списка: пара пустых скобок и специальный символ `nil`. Независимо от того, как вы введете пустой список, он будет отображен как `nil`.

```
> ()
NIL
> nil
NIL
```

Перед () необязательно ставить кавычку, так как символ nil самовычисляем.

2.4. Операции со списками

Построение списков осуществляется с помощью функции cons. Если второй ее аргумент – список, она возвращает новый список с первым аргументом, добавленным в его начало:

```
> (cons 'a '(b c d))
(A B C D)
```

Список из одного элемента также может быть создан с помощью cons и пустого списка. Функция list, с которой мы уже познакомились, – всего лишь более удобный способ последовательного использования cons.

```
> (cons 'a (cons 'b nil))
(A B)
> (list 'a 'b)
(A B)
```

Простейшие функции для получения отдельных элементов списка – car и cdr.° Функция car служит для вывода первого элемента списка, а cdr – для вывода всех элементов, кроме первого:

```
> (car '(a b c))
A
> (cdr '(a b c))
(B C)
```

Используя комбинацию функций car и cdr, можно получить любой элемент списка. Например, чтобы получить третий элемент, нужно написать:

```
> (car (cdr (cdr '(a b c d))))
C
```

Однако намного проще достичь того же результата с помощью функции third:

```
> (third '(a b c d))
C
```

2.5. Истинность

В Common Lisp истинность по умолчанию представляется символом t. Как и nil, символ t является самовычисляемым. Например, функция listp возвращает истину, если ее аргумент – список:

```
> (listp '(a b c))
T
```

Функции, возвращающие логические значения «истина» либо «ложь», называются *предикатами*. В Common Lisp имена предикатов часто оканчиваются на «p».

Ложь в Common Lisp представляется с помощью nil, пустого списка. Применяя listp к аргументу, который не является списком, получим nil:

```
> (listp 27)
NIL
```

Поскольку nil имеет два значения в Common Lisp, функция null, имеющая истинное значение для пустого списка:

```
> (null nil)
T
```

и функция not, возвращающая истинное значение, если ее аргумент ложен:

```
> (not nil)
T
```

делают одно и то же.

Простейший условный оператор в Common Lisp – if. Обычно он принимает три аргумента: *test*-, *then*- и *else*-выражения. Сначала вычисляется *тестовое test*-выражение. Если оно истинно, вычисляется *then*-выражение (*«то»*) и возвращается его значение. В противном случае вычисляется *else*-выражение (*«иначе»*).

```
> (if (listp '(a b c))
      (+ 1 2)
      (+ 5 6))
3
> (if (listp 27)
      (+ 1 2)
      (+ 5 6))
11
```

Как и quote, if – это специальный оператор, а не функция, так как для функции вычисляются все аргументы, а у оператора if вычисляется лишь одно из двух последних выражений.

Указывать последний аргумент if необязательно. Если он пропущен, то автоматически принимается за nil.

```
> (if (listp 27)
      (+ 2 3))
NIL
```

Несмотря на то, что по умолчанию истина представляется в виде t, любое выражение, кроме nil, также считается истинным:

```
> (if 27 1 2)
1
```

Логические операторы and (и) и or (или) действуют похожим образом. Оба могут принимать любое количество аргументов, но вычисляют их до тех пор, пока не будет ясно, какое значение необходимо вернуть. Если все аргументы истинны (то есть не nil), то оператор and вернет значение последнего:

```
> (and t (+ 1 2))
3
```

Но если один из аргументов окажется ложным, то следующие за ним аргументы не будут вычислены. Так же действует и or, вычисляя значения аргументов до тех пор, пока среди них не найдется хотя бы одно истинное значение.

Эти два оператора – *макросы*. Как и специальные операторы, макросы могут обходить обычный порядок вычисления. В главе 10 объясняется, как писать собственные макросы.

2.6. Функции

Новые функции можно определить с помощью оператора defun. Он обычно принимает три или более аргументов: имя, список параметров и одно или более выражений, которые составляют тело функции. Вот как мы можем с помощью defun определить функцию third:

```
> (defun our-third (x)
    (car (cdr (cdr x))))
OUR-THIRD
```

Первый аргумент задает имя функции, в нашем примере это our-third. Второй аргумент, список (x), сообщает, что функция может принимать строго один аргумент: x. Используемый здесь символ x называется *переменной*. Когда переменная представляет собой аргумент функции, как x в этом примере, она еще называется *параметром*.

Оставшаяся часть, (car (cdr (cdr x))), называется *телом* функции. Она сообщает Лиспу, что нужно сделать, чтобы вернуть значение из функции. Вызов our-third возвращает (car (cdr (cdr x))), какое бы значение аргумента x ни было задано:

```
> (our-third '(a b c d))
C
```

Теперь, когда мы познакомились с переменными, будет легче понять, чем являются символы. Это попросту имена переменных, существующие сами по себе. Именно поэтому символы, как и списки, нуждаются в цитировании. Так же как «закавыченные» списки не будут восприняты как код, так и «закавыченные» символы не будут перепутаны с переменными.

Функцию можно рассматривать как обобщение Лисп-выражения. Следующее выражение проверяет, превосходит ли сумма чисел 1 и 4 число 3:

```
> (> (+ 1 4) 3)
T
```

Заменяя конкретные числа переменными, мы можем получить функцию, выполняющую ту же проверку, но уже для трех произвольных чисел:

```
> (defun sum-greater (x y z)
    (> (+ x y) z))
SUM-GREATER
> (sum-greater 1 4 3)
T
```

В Лиспе нет различий между программой, процедурой и функцией. Все это функции (да и сам Лисп по большей части состоит из функций). Не имеет смысла определять одну *главную* функцию[1], ведь любая функция может быть вызвана в toplevel. В числе прочего, это означает, что программу можно тестировать по маленьким кусочкам в процессе ее написания.

2.7. Рекурсия

Функции, рассмотренные в предыдущем разделе, вызывали другие функции, чтобы те выполнили часть работы за них. Например, `sum-greater` вызывала + и >. Функция может вызывать любую другую функцию, включая саму себя.

Функции, вызывающие сами себя, называются *рекурсивными*. В Common Lisp есть функция `member`, которая проверяет, есть ли в списке какой-либо объект. Ниже приведена ее упрощенная реализация:

```
(defun our-member (obj lst)
  (if (null lst)
      nil
      (if (eql (car lst) obj)
          lst
          (our-member obj (cdr lst)))))
```

Предикат `eql` проверяет два аргумента на идентичность. Все остальное в этом выражении вам должно быть уже знакомо.

```
> (our-member 'b '(a b c))
(B C)
> (our-member 'z '(a b c))
NIL
```

Опишем словами, что делает эта функция. Чтобы проверить, есть ли `obj` в списке `lst`, мы:

---

[1] Как, например, в C, где требуется введение основной функции `main`. – *Прим. перев.*

1. Проверяем, пуст ли список `lst`. Если он пуст, значит, `obj` не присутствует в списке.
2. Если `obj` является первым элементом `lst`, значит, он есть в этом списке.
3. В другом случае проверяем, есть ли `obj` среди оставшихся элементов списка `lst`.

Подобное описание порядка действий будет полезным, если вы захотите понять, как работает рекурсивная функция.

Многим поначалу бывает сложно понять рекурсию. Причина этому – использование ошибочной метафоры для функций. Часто считается, что функция – это особый агрегат, который получает параметры в качестве сырья, перепоручает часть работы другим функциям-агрегатам и в итоге получает готовый продукт – возвращаемое значение. При таком рассмотрении возникает парадокс: как агрегат может перепоручать работу сам себе, если он уже занят?

Более подходящее сравнение для функции – процесс. Для процесса рекурсия вполне естественна. В повседневной жизни мы часто наблюдаем рекурсивные процессы и не задумываемся об этом. К примеру, представим себе историка, которому интересна динамика численности населения в Европе. Процесс изучения им соответствующих документов будет выглядеть следующим образом:

1. Получить копию документа.
2. Найти в нем информацию об изменении численности населения.
3. Если в нем также упоминаются иные источники, которые могут быть полезны, также изучить и их.

Этот процесс довольно легко понять, несмотря на то, что он рекурсивен, поскольку его третий шаг может повлечь за собой точно такой же процесс.

Поэтому не стоит расценивать функцию `our-member` как некий агрегат, который проверяет присутствие какого-либо элемента в списке. Это всего лишь набор правил, позволяющих выяснить этот факт. Если мы будем воспринимать функции подобным образом, то парадокс, связанный с рекурсией, исчезает.°

2.8. Чтение Лиспа

Определенная в предыдущем разделе функция заканчивается пятью закрывающими скобками. Более сложные функции могут заканчиваться семью-восемью скобками. Это часто обескураживает людей, которые только начинают изучать Лисп. Как можно читать подобный код, не говоря уже о том, чтобы писать его самому? Как искать в нем парные скобки?

Ответ прост: вам не нужно делать это. Лисп-программисты пишут и читают код, ориентируясь по отступам, а не по скобкам. К тому же любой

хороший текстовый редактор, особенно если он поставляется с Лисп-системой, умеет выделять парные скобки. Если ваш редактор не делает этого, остановитесь и постарайтесь узнать, как включить подобную функцию, поскольку без нее писать Лисп-код практически невозможно.[1]

С хорошим редактором поиск парных скобок в процессе написания кода перестанет быть проблемой. Кроме того, благодаря общепринятым соглашениям касательно выравнивания кода вы можете легко читать код, не обращая внимания на скобки.

Любой Лисп-хакер, независимо от своего стажа, с трудом разберет определение `our-member`, если оно будет записано в таком виде:

```
(defun our-member (obj lst) (if (null lst) nil (if (eql (car lst) obj) lst
(our-member obj (cdr lst)))))
```

С другой стороны, правильно выравненный код будет легко читаться даже без скобок:

```
defun our-member (obj lst)
  if (null lst)
     nil
     if eql (car lst) obj
        lst
        our-member obj (cdr lst)
```

Это действительно удобный подход при написании кода на бумаге, в то время как в редакторе вы сможете воспользоваться удобной возможностью поиска парных скобок.

2.9. Ввод и вывод

До сих пор мы осуществляли ввод-вывод с помощью toplevel. Чтобы ваша программа была по-настоящему интерактивной, этого явно недостаточно. В этом разделе мы рассмотрим несколько функций ввода-вывода.

Основная функция вывода в Common Lisp – `format`. Она принимает два или более аргументов: первый определяет, куда будет напечатан результат, второй – это строковый шаблон, а остальные аргументы – это объекты, которые будут вставляться в нужные позиции заданного шаблона. Вот типичный пример:

```
> (format t "~A plus ~A equals ~A.~%" 2 3 (+ 2 3))
2 plus 3 equals 5.
NIL
```

Обратите внимание, что здесь отображены два значения. Первая строка – результат выполнения `format`, напечатанный в toplevel. Обычно функции типа `format` не вызываются напрямую в toplevel, а используются внутри программ, поэтому возвращаемые ими значения не отображаются.

---

[1] В редакторе vi эта опция включается командой `:set sm`. В Emacs `M-x lisp-mode` будет отличным выбором.

Первый аргумент для функции format, t, показывает, что вывод будет отправлен в стандартное место, принятое по умолчанию. Обычно это toplevel. Второй аргумент – это строка, служащая шаблоном для вывода. В ней каждое ~A определяет позицию для заполнения, а символ ~% соответствует переносу строки. Позиции по порядку заполняются значениями оставшихся аргументов.

Стандартная функция чтения – read (ее также называют считывателем). Вызванная без аргументов, она выполняет чтение из стандартного места, которым обычно бывает toplevel. Ниже приведена функция, которая предлагает ввести любое значение и затем возвращает его:

```
(defun askem (string)
  (format t "~A" string)
  (read))
```

Это работает так:

```
> (askem "How old are you? ")
How old are you? 29
29
```

Учтите, что для завершения считывания read будет ожидать нажатия вами Enter. Поэтому не стоит использовать функцию read, не печатая перед этим приглашение ко вводу, иначе может показаться, что программа зависла, хотя на самом деле она просто ожидает ввода.

Другая вещь, которую следует знать о функции read: это поистине мощный инструмент. Фактически это полноценный обработчик Лисп-выражений. Она не просто читает символы и возвращает их в виде строки – она обрабатывает введенное выражение и возвращает полученный лисповый объект. В приведенном выше примере функция read возвращает число.

Несмотря на свою краткость, определение askem содержит нечто, чего мы до сих пор не встречали – тело функции состоит из нескольких выражений. Вообще говоря, оно может содержать любое количество выражений. При вызове функции они вычисляются последовательно, и возвращается значение последнего выражения.

Во всех предыдущих разделах мы придерживались «чистого» Лиспа, то есть Лиспа без *побочных эффектов*. Побочный эффект – это событие, которое каким-либо образом изменяет состояние системы. При вычислении выражения (+ 1 2) возвращается значение 3, и никаких побочных эффектов не происходит. В последнем же примере побочный эффект производится функцией format, которая не только возвращает значение, но и печатает что-то еще. Это одна из разновидностей побочных эффектов.

Если мы зададимся целью писать код без побочных эффектов, то будет бессмысленно определять функции, состоящие из нескольких выражений, так как в качестве значения функции будет использоваться значение последнего аргумента, а значения предыдущих выражений будут

утеряны. Если эти выражения не будут вызывать побочных эффектов, то вы даже не узнаете, вычислял ли их Лисп вообще.

2.10. Переменные

Один из наиболее часто используемых операторов в Common Lisp – это `let`, который позволяет вам ввести новые локальные переменные:

```
> (let ((x 1) (y 2))
    (+ x y))
3
```

Выражение с использованием `let` состоит из двух частей. Первая содержит инструкции, определяющие новые переменные. Каждая такая инструкция содержит имя переменной и соответствующее ей выражение. Чуть выше мы создали две новые переменные, `x` и `y`, которым были присвоены значения `1` и `2`. Эти переменные действительны внутри тела `let`.

За списком переменных и значений следуют выражения, которые вычисляются по порядку. В нашем случае имеется только одно выражение, `(+ x y)`. Значением всего вызова `let` будет значение последнего выражения. Давайте напишем более избирательный вариант функции `askem` с использованием `let`:

```
(defun ask-number ()
  (format t "Please enter a number. ")
  (let ((val (read)))
    (if (numberp val)
        val
        (ask-number))))
```

Мы создаем переменную `val`, содержащую результат вызова `read`. Сохраняя это значение, мы можем проверить, что было прочитано. Как вы уже догадались, `numberp` – это предикат, проверяющий, является ли его аргумент числом.

Если введенное значение – нечисло, то `ask-number` вызовет саму себя, чтобы пользователь повторил попытку. Так будет повторяться до тех пор, пока функция `read` не получит число:

```
> (ask-number)
Please enter a number. a
Please enter a number. (no hum)
Please enter a number. 52
52
```

Переменные типа `val`, создаваемые оператором `let`, называются *локальными*, то есть действительными в определенной области. Есть также *глобальные* переменные, которые действительны везде.[1]

[1] Реальная разница между локальными и глобальными переменными будет пояснена в главе 6.

Глобальная переменная может быть создана с помощью оператора defparameter:

```
> (defparameter *glob* 99)
*GLOB*
```

Такая переменная будет доступна везде, кроме выражений, в которых создается локальная переменная с таким же именем. Чтобы избегать таких случайных совпадений, принято давать глобальным переменным имена, окруженные звездочками.

Также в глобальном окружении можно задавать константы, используя оператор defconstant[1]:

```
(defconstant limit (+ *glob* 1))
```

Нет необходимости давать константам имена, окруженные звездочками, потому что любая попытка определить переменную с таким же именем закончится ошибкой. Чтобы узнать, соответствует ли какое-то имя глобальной переменной или константе, воспользуйтесь boundp:

```
> (boundp '*glob*)
T
```

2.11. Присваивание

В Common Lisp наиболее общим средством присваивания значений является оператор setf. Он может присваивать значения переменным любых типов:

```
> (setf *glob* 98)
98
> (let ((n 10))
    (setf n 2)
    n)
2
```

Если setf пытается присвоить значение переменной, которая в данном контексте не является локальной, то переменная будет определена как глобальная:

```
> (setf x (list 'a 'b 'c))
(A B C)
```

Таким образом, вы можете создавать глобальные переменные неявно, просто присваивая им значения. Однако в файлах исходного кода считается хорошим тоном задавать их явно с помощью оператора defparameter.

---

[1] Как и для специальных переменных, для определяемых пользователем констант существует негласное правило, согласно которому их имена следует окружать знаками +. Следуя этому правилу, в примере выше мы бы определили константу +limit+. – *Прим. перев.*

Но можно делать больше, чем просто присваивать значения переменным. Первым аргументом setf может быть не только переменная, но и выражение. В таком случае значение второго аргумента вставляется в то *место*, на которое ссылается это выражение:

```
> (setf (car x) 'n)
N
> x
(N B C)
```

Первым аргументом setf может быть почти любое выражение, которое ссылается на какое-то место. Все такие операторы отмечены в приложении D как работающие с setf.

Также setf можно передать любое (четное) число аргументов. Следующий вызов

```
(setf a b
      c d
      e f)
```

эквивалентен трем последовательным вызовам setf:

```
(setf a b)
(setf c d)
(setf e f)
```

2.12. Функциональное программирование

Функциональное программирование – это понятие, которое означает написание программ, работающих через возвращаемые значения, а не изменения среды. Это основная парадигма в Лиспе. Большинство встроенных в Лисп функций не вызывают побочных эффектов.

Например, функция remove принимает список и объект и возвращает новый список, состоящий из тех же элементов, что и предыдущий, за исключением этого объекта.

```
> (setf lst '(c a r a t))
(C A R A T)
> (remove 'a lst)
(C R T)
```

Почему бы не сказать, что remove просто удаляет элемент из списка? Потому что она этого не делает. Исходный список остался нетронутым:

```
> lst
(C A R A T)
```

А что если вы хотите действительно удалить элементы из списка? В Лиспе принято предоставлять список в качестве аргумента какой-либо функции и присваивать возвращаемое ей значение с помощью setf. Чтобы удалить все a из списка x, мы скажем:

```
(setf x (remove 'a x))
```

В функциональном программировании избегают использовать setf и другие подобные вещи. Поначалу сложно вообразить, что это попросту возможно, не говоря уже о том, что желательно. Как можно создавать программы только с помощью возвращения значений?

Конечно, совсем без побочных эффектов работать неудобно. Однако, читая дальше эту книгу, вы будете все сильнее удивляться тому, как, в сущности, немного побочных эффектов необходимо. И чем меньше вы будете ими пользоваться, тем лучше.

Одно из главных преимуществ использования функционального программирования заключается в *интерактивном тестировании*. Это означает, что вы можете тестировать функции, написанные в функциональном стиле, сразу же после их написания, и при этом вы будете уверены, что результат выполнения всегда будет одинаковым. Вы можете изменить одну часть программы, и это не повлияет на корректность работы другой ее части. Такая возможность безбоязненно изменять код делает доступным новый стиль программирования, подобно тому как телефон в сравнении с письмами предоставляет новый способ общения.

2.13. Итерация

Когда мы хотим повторить что-то многократно, порой удобнее использовать итерацию, чем рекурсию. Типичный пример использования итерации – генерация таблиц. Следующая функция

```
(defun show-squares (start end)
  (do ((i start (+ i 1)))
      ((> i end) 'done)
    (format t "~A ~A~%" i (* i i))))
```

печатает квадраты целых чисел от start до end:

```
> (show-squares 2 5)
2 4
3 9
4 16
5 25
DONE
```

Макрос do – основной итерационный оператор в Common Lisp. Как и в случае let, первый аргумент do задает переменные. Каждый элемент этого списка может выглядеть как:

(*variable initial update*)

где *variable* – символ, а *initial* и *update* – выражения. Исходное значение каждой переменной определяется значением *initial*, и на каждой итерации это значение будет меняться в соответствии со значением *update*. В примере show-squares do использует только одну переменную, i. На первой итерации значение i равно start, и после каждой успешной итерации оно будет увеличиваться на единицу.

Второй аргумент do – список, состоящий по меньшей мере из одного выражения, которое определяет, когда итерации следует остановиться. В нашем примере этим выражением является проверка (> i end). Остальные выражения в списке будут вычислены после завершения итераций, и значение последнего из них будет возвращено как значение оператора do. Функция show-squares всегда возвращает done.

Оставшиеся аргументы do составляют тело цикла. Они будут вычисляться по порядку на каждой итерации. На каждом шаге в первую очередь модифицируются значения переменных, затем следует проверка условия окончания цикла, после которой (если условие не выполнено) вычисляется тело цикла.

Для сравнения запишем рекурсивную версию show-squares:

```
(defun show-squares (i end)
  (if (> i end)
      'done
      (progn
        (format t "~A ~A~%" i (* i i))
        (show-squares (+ i 1) end))))
```

В ней мы использовали новый оператор progn. Он принимает любое количество выражений, вычисляет их одно за другим и возвращает значение последнего.

Для частных случаев в Common Lisp существуют более простые итерационные операторы. К примеру, запишем функцию, определяющую длину списка:

```
(defun our-length (lst)
  (let ((len 0))
    (dolist (obj lst)
      (setf len (+ len 1)))
    len))
```

В этом примере оператор dolist принимает первый аргумент вида (*variable expression*) и следующий за ним набор выражений. Они вычисляются для каждого элемента списка, значение которых по очереди присваивается переменной. В этом примере для каждого obj в lst увеличивается значение len.

Рекурсивная версия этой функции:

```
(defun our-length (lst)
  (if (null lst)
      0
      (+ (our-length (cdr lst)) 1)))
```

Если список пуст, его длина равна 0, иначе она на 1 больше длины списка cdr. Эта версия функции our-length более понятна, однако она неэффективна, поскольку не использует хвостовую рекурсию (см. раздел 13.2).

2.14. Функции как объекты

В Лиспе функции – это самые обычные объекты, такие же как символы, строки или списки. Давая функции имя с помощью function, мы получаем ассоциированный объект. Как и quote, function – это специальный оператор, и поэтому нам не нужно брать в кавычки его аргумент:

```
> (function +)
#<Compiled-Function + 17BA4E>
```

Таким странным образом отображаются функции в типичной реализации Common Lisp.

До сих пор мы имели дело только с такими объектами, которые при печати отображаются так же, как мы их ввели. Это соглашение не распространяется на функции. Встроенная функция + обычно является куском машинного кода. В каждой реализации Common Lisp может быть свой способ отображения функций.

Аналогично использованию кавычки вместо quote возможно употребление #' как сокращения для function:

```
> #'+
#<Compiled-Function + 17BA4E>
```

Как и любой другой объект, функция может служить аргументом. Примером функции, аргументом которой является функция, является apply. Она требует функцию и список ее аргументов и возвращает результат вызова этой функции с заданными аргументами:

```
> (apply #'+ '(1 2 3))
6
> (+ 1 2 3)
6
```

Apply принимает любое количество аргументов, но последний из них обязательно должен быть списком:

```
> (apply #'+ 1 2 '(3 4 5))
15
```

Функция funcall делает то же самое, но не требует, чтобы аргументы были упакованы в список:

```
> (funcall #'+ 1 2 3)
6
```

Обычно создание функции и определение ее имени осуществляется с помощью макроса defun. Но функция не обязательно должна иметь имя, и для ее определения мы не обязаны использовать defun. Подобно большинству других объектов Лиспа, мы можем задавать функции буквально.

Чтобы буквально сослаться на число, мы используем последовательность цифр. Чтобы таким же образом сослаться на функцию, мы используем

лямбда-выражение. Лямбда-выражение – это список, содержащий символ `lambda` и следующие за ним список аргументов и *тело*, состоящее из 0 или более выражений. Ниже приведено лямбда-выражение, представляющее функцию, которая складывает два числа и возвращает их сумму:

```
(lambda (x y)
  (+ x y))
```

Список `(x y)` содержит параметры, за ним следует тело функции.

Лямбда-выражение можно считать именем функции. Как и обычное имя функции, лямбда-выражение может быть первым элементом вызова функции:

```
> ((lambda (x) (+ x 100)) 1)
101
```

а добавляя `#'` к этому выражению, можно получить соответствующую функцию:

```
> (funcall #'(lambda (x) (+ x 100))
           1)
101
```

Помимо прочего, такая запись позволяет использовать функции, не присваивая им имена.

Что такое Лямбда?

В лямбда-выражении `lambda` не является оператором. Это просто символ.° В ранних диалектах Лиспа он имел свою цель: функции имели внутреннее представление в виде списков, и единственным способом отличить функцию от обычного списка была проверка того, является ли первый его элемент символом `lambda`.

В Common Lisp вы можете задать функцию в виде списка, но они будут иметь отличное от списка внутреннее представление, поэтому `lambda` больше не требуется. Было бы вполне возможно записывать функции, например, так:

```
((x) (+ x 100))
```

вместо

```
(lambda (x) (+ x 100))
```

но Лисп-программисты привыкли начинать функции символом `lambda`, и Common Lisp также следует этой традиции.

2.15. Типы

Лисп обладает необыкновенно гибкой системой типов. Во многих языках необходимо задать конкретный тип для каждой переменной перед ее использованием. В Common Lisp значения имеют типы, а переменные – нет. Представьте, что у каждого объекта есть метка, которая определяет его тип. Такой подход называется *динамической типизацией*. Вам не нужно объявлять типы переменных, поскольку любая переменная может содержать объект любого типа.

Несмотря на то, что объявления типов вовсе не обязательны, вы можете задавать их из соображений производительности. Объявления типов обсуждаются в разделе 13.3.

В Common Lisp встроенные типы образуют иерархию подтипов и надтипов. Объект всегда имеет несколько типов. Например, число `27` соответствует типам `fixnum`, `integer`, `rational`, `real`, `number`, `atom` и `t` в порядке увеличения общности. (Численные типы обсуждаются в главе 9.) Тип `t` является самым верхним во всей иерархии, поэтому каждый объект имеет тип `t`.

Функция `typep` определяет, принадлежит ли ее первый аргумент к типу, который задается вторым аргументом:

```
> (typep 27 'integer)
T
```

По мере изучения Common Lisp мы познакомимся с самыми разными встроенными типами.

2.16. Заглядывая вперед

В этой главе мы довольно поверхностно познакомились с Лиспом. Но все же начинает вырисовываться портрет этого довольно необычного языка. Начнем с того, что он имеет единый синтаксис для записи всего кода. Синтаксис основан на списках, которые являются разновидностью объектов Лиспа. Функции, которые также являются Лисп-объектами, могут быть представлены в виде списков. А сам Лисп – это тоже программа на Лиспе, почти полностью состоящая из встроенных Лисп-функций, которые не отличаются от тех функций, которые вы можете определить самостоятельно.

Не смущайтесь, если эта цепочка взаимоотношений пока вам не совсем ясна. Лисп содержит настолько много новых идей, что их осознание требует времени. Но одно должно быть ясно: некоторые из этих идей изначально довольно элегантны.

Ричард Гэбриэл однажды в полушуточной форме назвал C языком для написания UNIX.° Также и мы можем назвать Лисп языком для написания Лиспа. Но между этими утверждениями есть существенная разница. Язык, который легко написать на самом себе, кардинально отли-

чается от языка, годного для написания каких-то отдельных классов программ. Он открывает новый способ программирования: вы можете как писать программы на таком языке, так и совершенствовать язык в соответствии с требованиями своих программ. Если вы хотите понять суть программирования на Лиспе, начните с этой идеи.

Итоги главы

1. Лисп интерактивен. Вводя выражение в toplevel, вы тут же получаете его значение.
2. Программы на Лиспе состоят из выражений. Выражение может быть атомом или списком, состоящим из оператора и следующих за ним аргументов. Благодаря префиксной записи этих аргументов может быть сколько угодно.
3. Порядок вычисления вызовов функций в Common Lisp таков: сначала вычисляются аргументы слева направо, затем они передаются оператору. Оператор `quote` не подчиняется этому порядку и возвращает само выражение неизменным (вместо его значения).
4. Помимо привычных типов данных в Лиспе есть символы и списки. Программы на Лиспе записываются в виде списков, поэтому легко составлять программы, которые пишут другие программы.
5. Есть три основные функции для обращения со списками: `cons`, строящая список; `car`, возвращающая первый элемент списка; `cdr`, возвращающая весь список, за исключением его первого элемента.
6. В Common Lisp символ `t` имеет истинное значение, `nil` – ложное. В контексте логических операций все, кроме `nil`, является истинным. Основной условный оператор – `if`. Операторы `and` и `or` также могут считаться условными.
7. Лисп состоит по большей части из функций. Собственные функции можно создавать с помощью `defun`.
8. Функция, которая вызывает сама себя, называется рекурсивной. Рекурсивную функцию следует понимать как процесс, но не как агрегат.
9. Скобки не создают затруднений, поскольку программисты пишут и читают Лисп-код по отступам.
10. Основные функции ввода-вывода: `read`, являющаяся полноценным обработчиком Лисп-выражений, и `format`, использующая для вывода заданный шаблон.
11. Локальные переменные создаются с помощью `let`, глобальные – с помощью `defparameter`.
12. Для присваивания используется оператор `setf`. Его первый аргумент может быть как переменной, так и выражением.

13. Основной парадигмой Лиспа является функциональное программирование, то есть написание кода без побочных эффектов.
14. Основной итерационный оператор – `do`.
15. Функции являются обычными объектами в Лиспе. Они могут записываться в виде лямбда-выражений, а также использоваться как аргументы для других функций.
16. В Лиспе значения имеют типы, а переменные – нет.

Упражнения

1. Опишите, что происходит при вычислении следующих выражений:

   (a) `(+ (- 5 1) (+ 3 7))`
   (b) `(list 1 (+ 2 3))`
   (c) `(if (listp 1) (+ 1 2) (+ 3 4))`
   (d) `(list (and (listp 3) t) (+ 1 2))`

2. Составьте с помощью `cons` три различных выражения, создающие список `(a b c)`.
3. С помощью `car` и `cdr` определите функцию, возвращающую четвертый элемент списка.
4. Определите функцию, принимающую два аргумента и возвращающую наибольший.
5. Что делают следующие функции?

   (a)
   ```
   (defun enigma (x)
     (and (not (null x))
          (or (null (car x))
              (enigma (cdr x)))))
   ```

   (b)
   ```
   (defun mystery (x y)
     (if (null y)
         nil
         (if (eql (car y) x)
             0
             (let ((z (mystery x (cdr y))))
               (and z (+ z 1))))))
   ```

6. Что может стоять на месте `x` в следующих выражениях?

   (a)
   ```
   > (car (x (cdr '(a (b c) d))))
   B
   ```

   (b)
   ```
   > (x 13 (/ 1 0))
   13
   ```

   (c)
   ```
   > (x #'list 1 nil)
   (1)
   ```

7. Определите функцию, проверяющую, является ли списком хотя бы один элемент списка. Пользуйтесь только теми операторами, которые были упомянуты в этой главе.
8. Предложите итеративное и рекурсивное определение функции, которая:
   (a) печатает количество точек, которое равно заданному положительному целому числу;
   (b) возвращает количество символов `a` в заданном списке.
9. Ваш товарищ пытается написать функцию, которая суммирует все значения элементов списка, кроме `nil`. Он написал две версии такой функции, но ни одна из них не работает. Объясните, что не так в каждой из них, и предложите корректную версию:

   (a)
```
(defun summit (lst)
  (remove nil lst)
  (apply #'+ lst))
```

   (b)
```
(defun summit (lst)
  (let ((x (car lst)))
    (if (null x)
        (summit (cdr lst))
        (+ x (summit (cdr lst))))))
```

3

Списки

Списки – это одна из базовых структур данных в Лиспе. В ранних диалектах списки были единственной структурой данных, и именно им Лисп обязан своим названием: «LISt Processor» (Обработчик списков). Нынешний Лисп уже не соответствует этому акрониму. Common Lisp является языком общего назначения и предоставляет программисту широкий набор структур данных.

Процесс разработки Лисп-программ часто напоминает эволюцию самого Лиспа. В первоначальной версии программы вы можете использовать множество списков, однако в ее более поздних версиях целесообразнее будет перейти к использованию более специализированных и эффективных структур данных. В этой главе подробно описываются операции со списками, а также с их помощью поясняются некоторые общие концепции Лиспа.

3.1. Ячейки

В разделе 2.4 вводятся `cons`, `car` и `cdr` – простейшие функции для манипуляций со списками. В действительности, `cons` объединяет два объекта в один, называемый *ячейкой* (*cons*). Если быть точнее, то `cons` – это пара указателей, первый из которых указывает на `car`, второй – на `cdr`.

С помощью `cons`-ячеек удобно объединять в пару объекты любых типов, в том числе и другие ячейки. Именно благодаря такой возможности с помощью `cons` можно строить произвольные списки.

Незачем представлять каждый список в виде `cons`-ячеек, но нужно помнить, что они могут быть заданы таким образом. Любой непустой список может считаться парой, содержащей первый элемент списка и остальную его часть. В Лиспе списки являются воплощением этой идеи. Именно поэтому функция `car` позволяет получить первый элемент списка,

а cdr – его остаток (который является либо cons-ячейкой, либо nil). И договоренность всегда была таковой: использовать car для обозначения первого элемента списка, а cdr – для его остатка. Так эти названия стали синонимами операций first и rest. Таким образом, списки – это не отдельный вид объектов, а всего лишь набор связанных между собой cons-ячеек.

Если мы попытаемся использовать cons вместе с nil,

```
> (setf x (cons 'a nil))
(A)
```

то получим список, состоящий из одной ячейки, как показано на рис. 3.1. Такой способ изображения ячеек называется *блочным*, потому что каждая ячейка представляется в виде блока, содержащего указатели на car и cdr. Вызывая car или cdr, мы получаем объект, на который указывает соответствующий указатель:

```
> (car x)
A
> (cdr x)
NIL
```

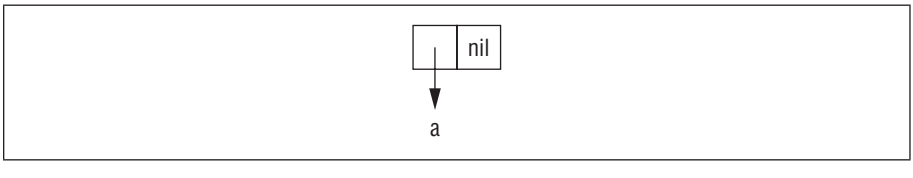

Рис. 3.1. *Список, состоящий из одной ячейки*

Составляя список из нескольких элементов, мы получаем цепочку ячеек:

```
> (setf y (list 'a 'b 'c))
(A B C)
```

Эта структура показана на рис. 3.2. Теперь cdr списка будет указывать на список из двух элементов:

```
> (cdr y)
(B C)
```

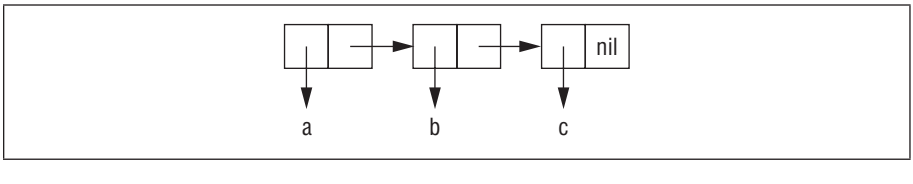

Рис. 3.2. *Список из трех ячеек*

Для списка из нескольких элементов указатель на car дает первый элемент списка, а указатель на cdr – его остаток.

Элементами списка могут быть любые объекты, в том числе и другие списки:

```
> (setf z (list 'a (list 'b 'c) 'd))
(A (B C) D)
```

Соответствующая структура показана на рис. 3.3; car второй ячейки указывает на другой список:

```
> (car (cdr z))
(B C)
```

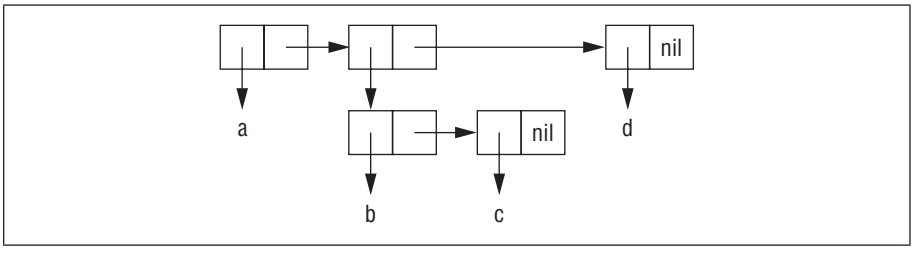

Рис. 3.3. *Вложенный список*

В этих двух примерах списки состояли из трех элементов. При этом в последнем примере один из элементов тоже является списком. Такие списки называют *вложенными* (*nested*), в то время как списки, не содержащие внутри себя подсписков, называют *плоскими* (*flat*).

Проверить, является ли объект cons-ячейкой, можно с помощью функции consp. Поэтому listp можно определить так:

```
(defun out-listp (x)
  (or (null x) (consp x)))
```

Теперь определим предикат atom, учитывая тот факт, что атомами в Лиспе считается все, кроме cons-ячеек:

```
(defun our-atom (x) (not (consp x)))
```

Исключением из этого правила считается nil, который является и атомом, и списком одновременно.

3.2. Равенство

Каждый раз, когда вы вызываете функцию cons, Лисп выделяет память для двух указателей. Это означает, что, вызывая cons дважды с одними и теми же аргументами, мы получим два значения, которые будут выглядеть идентично, но соответствовать разным объектам:

```
> (eql (cons 'a nil) (cons 'a nil))
NIL
```

Функция eql[1] возвращает t (true), только если сравниваемые значения соответствуют одному объекту в памяти Лиспа.

```
> (setf x (cons 'a nil))
(A)
> (eql x x)
T
```

Для проверки идентичности списков (и других объектов) используется предикат equal. С его точки зрения два одинаково выглядящих объекта равны:

```
> (equal x (cons 'a nil))
T
```

Покажем, как можно определить функцию equal для частного случая проверки на равенство списков[2], предполагая, что если два объекта равны для предиката eql, то они будут равны и для equal:

```
(defun our-equal (x y)
  (or (eql x y)
      (and (consp x)
           (consp y)
           (our-equal (car x) (car y))
           (our-equal (cdr x) (cdr y)))))
```

3.3. Почему в Лиспе нет указателей

Чтобы понять, как устроен Лисп, необходимо осознать, что механизм присваивания значения переменным похож на построение списков из объектов. Переменной соответствует указатель на ее значение, так же как cons-ячейки имеют указатели на car и cdr.

Некоторые другие языки позволяют оперировать указателями явно. Лисп же выполняет всю работу с указателями самостоятельно. Мы уже видели, как это происходит, на примере списков. Нечто похожее происходит и с переменными. Пусть две переменные указывают на один и тот же список:

```
> (setf x '(a b c))
(A B C)
> (setf y x)
(A B C)
```

---

[1] В ранних диалектах Лиспа задачу eql выполнял eq. В Common Lisp eq – более строгая функция, а основным предикатом проверки идентичности является eql. Роль eq разъясняется на стр. 234.

[2] Функция our-equal применима не к любым спискам, а только к спискам символов. Неточность в оригинале указана Биллом Стретфордом. – *Прим. перев.*

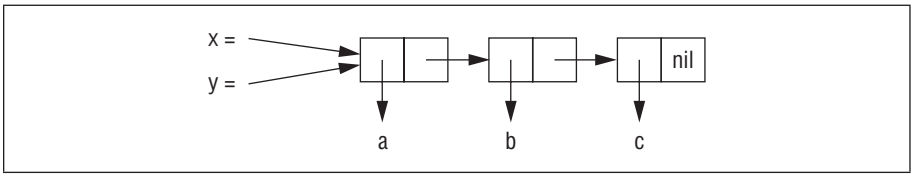

Рис. 3.4. *Две переменные, указывающие на один список*

Что происходит, когда мы пытаемся присвоить `y` значение `x`? Место в памяти, связанное с переменной `x`, содержит не сам список, а указатель на него. Чтобы присвоить переменной `y` то же значение, достаточно просто скопировать этот указатель (рис. 3.4). В данном случае две переменные будут одинаковыми с точки зрения `eql`:

```
> (eql x y)
T
```

Таким образом, в Лиспе указатели явно не используются, потому что любое значение, по сути, является указателем. Когда вы присваиваете значение переменной или сохраняете его в какую-либо структуру данных, туда, на самом деле, записывается указатель. Когда вы запрашиваете содержимое какой-либо структуры данных или значение переменной, Лисп возвращает данные, на которые ссылается указатель. Но это происходит неявно, поэтому вы можете записывать значения в структуры или «в» переменные, не задумываясь о том, как это происходит.

Из соображений производительности Лисп взамен указателей иногда использует непосредственное представление данных. Например, небольшие целые числа занимают не больше места, чем указатель, поэтому некоторые реализации Лиспа оперируют непосредственно целыми числами, а не указателями на них. Но для программиста важно лишь то, что в Лиспе по умолчанию можно положить что угодно куда угодно. И если вы явно не объявили обратного, то можно записать любой вид объекта в любую структуру данных, включая и саму структуру в себя.

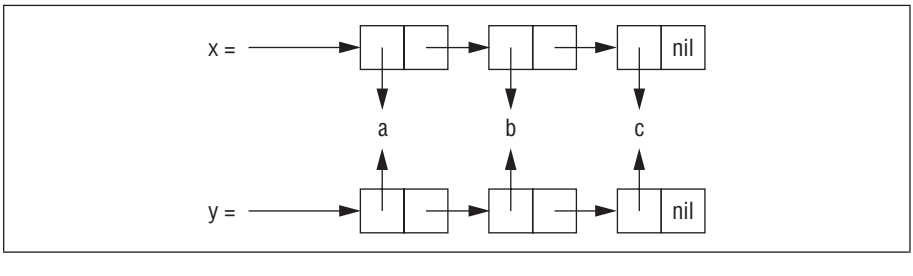

Рис. 3.5. *Результат копирования*

3.4. Построение списков

Функция copy-list принимает список и возвращает его копию. Новый список будет иметь те же элементы, но они будут размещены в других cons-ячейках:

```
> (setf x '(a b c)
        y (copy-list x))
(A B C)
```

Такая структура ячеек показана на рис. 3.5. Собственную функцию copy-list определим так:

```
(defun our-copy-list (lst)
 (if (atom lst)
     lst
     (cons (car lst) (our-copy-list (cdr lst)))))
```

Здесь подразумевается, что x и (copy-list x) всегда равны с точки зрения equal, а для eql равны, только если x является nil.

Функция append – еще одна функция для работы со списками, склеивающая между собой произвольное количество списков:

```
> (append '(a b) '(c d) '(e))
(A B C D E)
```

Функция append копирует все аргументы, за исключением последнего.

3.5. Пример: сжатие

В этом разделе приводится пример разработки простого механизма сжатия списков. Рассматриваемый ниже алгоритм принято называть *кодированием повторов* (*run-length encoding*). Представьте ситуацию: в ресторане официант обслуживает столик, за которым сидят четыре посетителя:

> «Что вы будете заказывать?» – спрашивает он.
>
> «Принесите фирменное блюдо, пожалуйста», – отвечает первый посетитель.
>
> «И мне тоже», – говорит второй.
>
> «Ну и я за компанию», – присоединяется третий.
>
> Все смотрят на четвертого клиента. «А я бы предпочел кориандровое суфле», – тихо произносит он.
>
> Официант со вздохом разворачивается и идет к кухне. «Три фирменных, – кричит он повару, – и одно кориандровое суфле».

На рис. 3.6 показан подобный алгоритм для списков. Функция compress принимает список из атомов и возвращает его сжатое представление:

```
> (compress '(1 1 1 0 1 0 0 0 0 1))
((3 1) 0 1 (4 0) 1)
```

```
(defun compress (x)
  (if (consp x)
      (compr (car x) 1 (cdr x))
      x))

(defun compr (elt n lst)
  (if (null lst)
      (list (n-elts elt n))
      (let ((next (car lst)))
        (if (eql next elt)
            (compr elt (+ n 1) (cdr lst))
            (cons (n-elts elt n)
                  (compr next 1 (cdr lst)))))))

(defun n-elts (elt n)
  (if (> n 1)
      (list n elt)
      elt))
```

Рис. 3.6. *Кодирование повторов: сжатие*

Если какой-либо элемент повторяется несколько раз, он заменяется на список, содержащий этот элемент и число его повторений.

Большая часть работы выполняется рекурсивной функцией `compr`, которая принимает три аргумента: `elt` – последний встреченный элемент; `n` – число его повторений; `lst` – остаток списка, подлежащий дальнейшей компрессии. Когда список заканчивается, вызывается функция `n-elts`, возвращающая сжатое представление `n` элементов `elt`. Если первый элемент `lst` по-прежнему равен `elt`, увеличиваем `n` и идем дальше. В противном случае мы получаем сжатое представление предыдущей серии одинаковых элементов и присоединяем это к тому, что получим с помощью `compr` от остатка списка.

Чтобы реконструировать сжатый список, воспользуемся `uncompress` (рис. 3.7):

```
> (uncompress '((3 1) 0 1 (4 0) 1))
(1 1 1 0 1 0 0 0 0 1)
```

Эта функция выполняется рекурсивно, копируя атомы и раскрывая списки с помощью `list-of`:

```
> (list-of 3 'ho)
(HO HO HO)
```

В действительности, нет необходимости использовать `list-of`. Встроенная функция `make-list` выполняет ту же работу, однако использует аргументы по ключу (keyword), которых мы пока еще не касались.

Функции `compress` и `uncompress`, представленные на рис. 3.6 и 3.7, определены не так, как это сделал бы опытный программист. Они неэффек-

тивны, не осуществляют сжатие в достаточной мере и работают только со списками атомов. В следующих нескольких главах мы узнаем, как можно исправить все эти проблемы.

```
(defun uncompress (lst)
  (if (null lst)
      nil
      (let ((elt (car lst))
            (rest (uncompress (cdr lst))))
        (if (consp elt)
            (append (apply #'list-of elt)
                    rest)
            (cons elt rest)))))

(defun list-of (n elt)
  (if (zerop n)
      nil
      (cons elt (list-of (- n 1) elt))))
```

Рис. 3.7. *Кодирование повторов: декодирование*

Загрузка программ

Код в этом разделе впервые претендует на название отдельной программы. Если наша программа имеет достаточно большой размер, удобно сохранять ее текст в файл. Прочитать код из файла можно с помощью `load`. Если мы сохраним код с рис. 3.6 и 3.7 в файл под названием `"compress.lisp"`, то, набрав в toplevel

```
(load "compress.lisp")
```

получим такой же результат, как если бы набрали все выражения непосредственно в toplevel.

Учтите, что некоторые реализации Лиспа могут использовать расширение `".lsp"`, а не `".lisp"`.

3.6. Доступ

Для доступа к частям списков в Common Lisp имеются еще несколько функций, которые определяются с помощью `car` и `cdr`. Чтобы получить элемент с определенным индексом, вызовем функцию `nth`:

```
> (nth 0 '(a b c))
A
```

Чтобы получить n-й хвост списка, вызовем nthcdr:

```
> (nthcdr 2 '(a b c))
(C)
```

Функции nth и nthcdr ведут отсчет элементов списка с 0. Вообще говоря, в Common Lisp любая функция, обращающаяся к элементам структур данных, начинает отсчет с нуля.

Эти две функции очень похожи, и вызов nth эквивалентен вызову car от nthcdr. Определим nthcdr без обработки возможных ошибок:

```
(defun our-nthcdr (n lst)
  (if (zerop n)
      lst
      (our-nthcdr (- n 1) (cdr lst))))
```

Функция zerop всего лишь проверяет, равен ли нулю ее аргумент.

Функция last возвращает последнюю cons-ячейку списка:

```
> (last '(a b c))
(C)
```

Это не последний элемент; чтобы получить последний элемент, а не последнюю ячейку, воспользуйтесь функцией car от last.

В Common Lisp для доступа к элементам с первого по десятый выделены специальные функции, которые получили названия от английских порядковых числительных (от first до tenth). Обратите внимание, что отсчет начинается не с нуля и (second x) эквивалентен (nth 1 x).

Кроме того, в Common Lisp определены функции типа caddr (сокращенный вызов car от cdr от cdr). Также определены функции вида cxr, где x – набор всех возможных сочетаний a и d длиной до четырех символов. За исключением cadr, которая ссылается на второй элемент, не рекомендуется использовать подобные функции в коде, так как они затрудняют его чтение.

3.7. Отображающие функции

Common Lisp определяет несколько операций для применения какой-либо функции к каждому элементу списка. Чаще всего для этого используется mapcar, которая вызывает заданную функцию поэлементно для одного или нескольких списков и возвращает список результатов:

```
> (mapcar #'(lambda (x) (+ x 10))
          '(1 2 3))
(11 12 13)
> (mapcar #'list
          '(a b c)
          '(1 2 3 4))
((A 1) (B 2) (C 3))
```

Из последнего примера видно, как mapcar обрабатывает случай со списками разной длины. Вычисление обрывается по окончании самого короткого списка.

Похожим образом действует maplist, однако применяет функцию последовательно не к car, а к cdr списка, начиная со всего списка целиком.

```
> (maplist #'(lambda (x) x)
           '(a b c))
((A B C) (B C) (C))
```

Среди других отображающих функций можно отметить mapc, которая рассматривается на стр. 102, а также mapcan, с которой вы познакомитесь на стр. 209.

3.8. Деревья

Cons-ячейки также можно рассматривать как двоичные деревья: car соответствует правому поддереву, а cdr – левому. К примеру, список (a (b c) d) представлен в виде дерева на рис. 3.8. (Если повернуть его на 45° против часовой стрелки, он будет напоминать рис. 3.3.)

В Common Lisp есть несколько встроенных функций для работы с деревьями. Например, copy-tree принимает дерево и возвращает его копию. Определим аналогичную функцию самостоятельно:

```
(defun our-copy-tree (tr)
  (if (atom tr)
      tr
      (cons (our-copy-tree (car tr))
            (our-copy-tree (cdr tr)))))
```

Сравните ее с функцией copy-list (стр. 53). В то время как copy-list копирует только cdr списка, copy-tree копирует еще и car.

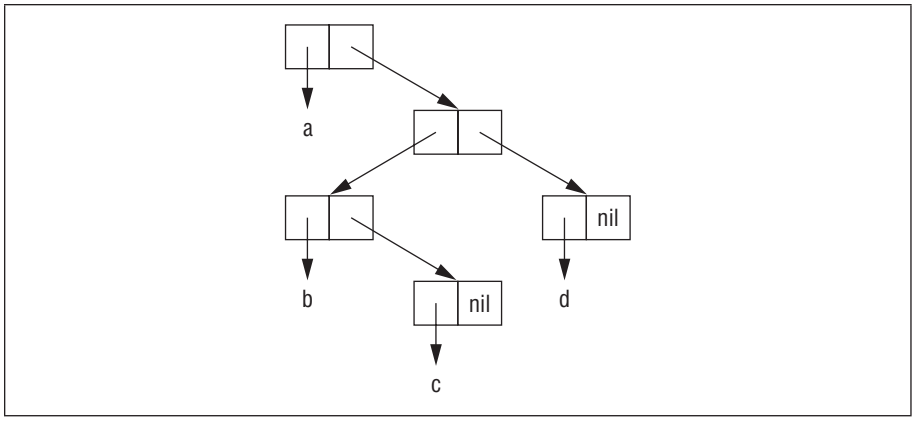

Рис. 3.8. Бинарное дерево

Бинарные деревья без внутренних узлов вряд ли окажутся полезными. Common Lisp включает в себя функции для операций с деревьями не потому, что без деревьев нельзя обойтись, а потому что эти функции очень полезны для работы со списками и подсписками. Например, предположим, что у нас есть список:

```
(and (integerp x) (zerop (mod x 2)))
```

И мы хотим заменить x на y. Заменить элементы в последовательности можно с помощью substitute:

```
> (substitute 'y 'x '(and (integerp x) (zerop (mod x 2))))
(AND (INTEGERP X) (ZEROP (MOD X 2)))
```

Как видите, использование substitute не дало результатов, так как список содержит три элемента, ни один из которых не является x. Здесь нам понадобится функция subst, работающая с деревьями:

```
> (subst 'y 'x '(and (integerp x) (zerop (mod x 2))))
(AND (INTEGERP Y) (ZEROP (MOD Y 2)))
```

Наше определение subst будет очень похоже на copy-tree:

```
(defun our-subst (new old tree)
  (if (eql tree old)
      new
      (if (atom tree)
          tree
          (cons (our-subst new old (car tree))
                (our-subst new old (cdr tree))))))
```

Любые функции, оперирующие с деревьями, будут выглядеть похожим образом, рекурсивно вызывая себя с car и cdr. Такая рекурсия называется *двойной*.

3.9. Чтобы понять рекурсию, нужно понять рекурсию

Чтобы понять, как работает рекурсия, студентам часто предлагают трассировать последовательность вызовов на бумаге. (Такая последовательность, например, изображена на стр. 291.) Иногда выполнение такого задания может ввести в заблуждение, так как программисту вовсе не обязательно четко представлять себе последовательность вызовов и возвращаемых результатов.

Если представлять рекурсию именно в таком виде, то ее применение вызовет лишь раздражение и вряд ли принесет пользу. Преимущества рекурсии открываются при более абстрактном взгляде на нее.

Чтобы убедиться, что рекурсия делает то, что мы думаем, достаточно спросить, покрывает ли она все варианты. Посмотрим, к примеру, на рекурсивную функцию для определения длины списка:

```
(defun len (lst)
  (if (null lst)
      0
      (+ (len (cdr lst)) 1)))
```

Можно убедиться в корректности функции, проверив две вещи[1]:

1. Она работает со списками нулевой длины, возвращая 0.
2. Если она работает со списками, длина которых равна n, то будет справедлива также и для списков длиной n+1.

Если оба случая верны, то функция ведет себя корректно на всех возможных списках.

Первое утверждение совершенно очевидно: если lst – это nil, то функция тут же возвращает 0. Теперь предположим, что она работает со списком длиной n. Согласно определению, для списка длиной n+1 она вернет число, на 1 большее длины cdr списка, то есть n+1.

Это все, что нам нужно знать. Представлять всю последовательность вызовов вовсе не обязательно, так же как необязательно искать парные скобки в определениях функций.

Для более сложных функций, например двойной рекурсии, случаев будет больше, но процедура останется прежней. К примеру, для функции our-copy-tree (стр. 41) потребуется рассмотреть три случая: атомы, простые ячейки, деревья, содержащие n+1 ячеек.

Первый случай носит название *базового* (*base case*). Если рекурсивная функция ведет себя не так, как ожидалось, причина часто заключается в некорректной проверке базового случая или же в отсутствии проверки, как в примере с функцией member:

```
(defun our-member (obj lst)             ; wrong[2]
  (if (eql (car lst) obj)
      lst
      (our-member obj (cdr lst))))
```

В этом определении необходима проверка списка на пустоту, иначе в случае отсутствия искомого элемента в списке рекурсивный вызов будет выполняться бесконечно. В приложении А эта проблема рассматривается более детально.

Способность оценить корректность рекурсивной функции – лишь первая часть понимания рекурсии. Вторая часть – необходимо научиться писать собственные рекурсивные функции, которые делают то, что вам требуется. Этому посвящен раздел 6.9.

---

[1] Такой метод доказательства носит название *математической индукции*. – *Прим. перев.*

[2] Здесь ; wrong – это *комментарий*. Комментариями в Лиспе считается все от символа «;» до конца текущей строки.

3.10. Множества

Списки – хороший способ представления небольших множеств. Чтобы проверить, принадлежит ли элемент множеству, задаваемому списком, можно воспользоваться функцией member:

```
> (member 'b '(a b c))
(B C)
```

Если искомый элемент найден, member возвращает не t, а часть списка, начинающегося с найденного элемента. Конечно, непустой список логически соответствует истине, но такое поведение member позволяет получить больше информации. По умолчанию member сравнивает аргументы с помощью eql. Предикат сравнения можно задать вручную с помощью аргумента по ключу. Аргументы по ключу (keyword) – довольно распространенный в Common Lisp способ передачи аргументов. Такие аргументы передаются не в соответствии с их положением в списке параметров, а с помощью особых меток, называемых ключевыми словами. Ключевым словом считается любой символ, начинающийся с двоеточия.

Одним из аргументов по ключу, принимаемых member, является :test. Он позволяет использовать в качестве предиката сравнения вместо eql произвольную функцию, например equal:

```
> (member '(a) '((a) (z)) :test #'equal)
((A) (Z))
```

Аргументы по ключу не являются обязательными и следуют последними в вызове функции, причем их порядок не имеет значения.

Другой аргумент по ключу функции member – :key. С его помощью можно задать функцию, применяемую к каждому элементу перед сравнением:

```
> (member 'a '((a b) (c d)) :key #'car)
((A B) (C D))
```

В этом примере мы искали элемент, car которого равен a.

При желании использовать оба аргумента по ключу можно задавать их в произвольном порядке:

```
> (member 2 '((1) (2)) :key #'car :test #'equal)
((2))
> (member 2 '((1) (2)) :test #'equal :key #'car)
((2))
```

С помощью member-if можно найти элемент, удовлетворяющий произвольному предикату, например oddp (истинному, когда аргумент нечетен):

```
> (member-if #'oddp '(2 3 4))
(3 4)
```

Наша собственная функция member-if могла бы выглядеть следующим образом:

```
(defun our-member-if (fn lst)
  (and (consp lst)
       (if (funcall fn (car lst)
           lst
           (our-member-if fn (cdr lst)))))
```

Функция adjoin – своего рода условный cons. Она присоединяет заданный элемент к списку, но только если его еще нет в этом списке (т. е. не member):

```
> (adjoin 'b '(a b c))
(A B C)
> (adjoin 'z '(a b c))
(Z A B C)
```

В общем случае adjoin принимает те же аргументы по ключу, что и member.

Common Lisp определяет основные логические операции с множествами, такие как объединение, пересечение, дополнение, для которых определены соответствующие функции: union, intersection, set-difference. Эти функции работают ровно с двумя списками и имеют те же аргументы по ключу, что и member.

```
> (union '(a b c) '(c b s))
(A C B S)
> (intersection '(a b c) '(b b c))
(B C)
> (set-difference '(a b c d e) '(b e))
(A C D)
```

Поскольку в множествах нет такого понятия, как упорядочение, эти функции не сохраняют порядок элементов в исходных списках. Например, вызов set-difference из примера может с тем же успехом вернуть (d c a).

3.11. Последовательности

Списки также можно рассматривать как последовательности элементов, следующих друг за другом в фиксированном порядке. В Common Lisp помимо списков к *последовательностям* также относятся векторы. В этом разделе вы научитесь работать со списками как с последовательностями. Более детально операции с последовательностями будут рассмотрены в разделе 4.4.

Длина последовательности определяется с помощью length:

```
> (length '(a b c))
3
```

Ранее мы писали урезанную версию этой функции, работающую только со списками.

Скопировать часть последовательности можно с помощью subseq. Второй аргумент (обязательный) задает начало подпоследовательности, а третий (необязательный) – индекс первого элемента, не подлежащего копированию.

```
> (subseq '(a b c d) 1 2)
(B)
> (subseq '(a b c d) 1)
(B C D)
```

Если третий аргумент пропущен, то подпоследовательность заканчивается вместе с исходной последовательностью.

Функция reverse возвращает последовательность, содержащую исходные элементы в обратном порядке:

```
> (reverse '(a b c))
(C B A)
```

С помощью reverse можно, например, искать *палиндромы*, т. е. последовательности, читаемые одинаково в прямом и обратном порядке (например, (a b b a)). Две половины палиндрома с четным количеством аргументов будут зеркальными отражениями друг друга. Используя length, subseq и reverse, определим функцию mirror?[1]:

```
(defun mirror? (s)
  (let ((len (length s)))
    (and (evenp len)
         (let ((mid (/ len 2)))
           (equal (subseq s 0 mid)
                  (reverse (subseq s mid)))))))
```

Эта функция определяет, является ли список таким палиндромом:

```
> (mirror? '(a b b a))
T
```

Для сортировки последовательностей в Common Lisp есть встроенная функция sort. Она принимает список, подлежащий сортировке, и функцию сравнения от двух аргументов:

```
> (sort '(0 2 1 3 8) #'>)
(8 3 2 1 0)
```

С функцией sort следует быть осторожными, потому что она *деструктивна*. Из соображений производительности sort не создает новый список, а модифицирует исходный. Поэтому если вы не хотите изменять исходную последовательность, передайте в функцию ее копию.°

Используя sort и nth, запишем функцию, которая принимает целое число n и возвращает n-й элемент в порядке убывания:

---

[1] Ричард Фейтман указал, что есть более простой способ проверки палиндрома, а именно: (equal x (reverse x)). – *Прим. перев.*

```
(defun nthmost (n lst)
  (nth (- n 1)
       (sort (copy-list lst) #'>)))
```

Мы вынуждены вычитать единицу, потому что nth индексирует элементы, начиная с нуля, а наша nthmost – с единицы.

```
> (nthmost 2 '(0 2 1 3 8))
3
```

Наша реализация nthmost не самая эффективная, но мы пока не будем вдаваться в тонкости ее оптимизации.

Функции every и some применяют предикат к одной или нескольким последовательностям. Если передана только одна последовательность, они проверяют, удовлетворяет ли каждый ее элемент этому предикату:

```
> (every #'oddp '(1 3 5))
T
> (some #'evenp '(1 2 3))
T
```

Если задано несколько последовательностей, предикат должен принимать количество аргументов, равное количеству последовательностей, и из каждой последовательности аргументы берутся по одному:

```
> (every #'> '(1 3 5) '(0 2 4))
T
```

Если последовательности имеют разную длину, кратчайшая из них ограничивает диапазон проверки.

3.12. Стопка

Представление списков в виде ячеек позволяет легко использовать их в качестве стопки (stack). В Common Lisp есть два макроса для работы со списком как со стопкой: (push x y) кладет объект x на вершину стопки y, (pop x) снимает со стопки верхний элемент. Оба эти макроса можно определить с помощью функции setf. Вызов

```
(push obj lst)
```

транслируется в

```
(setf lst (cons obj lst))
```

а вызов

```
(pop lst)
```

в

```
(let ((x (car lst))
  (setf lst (cdr lst))
  x)
```

Так, например

```
> (setf x '(b))
(B)
> (push 'a x)
(A B)
> x
(A B)
> (setf y x)
(A B)
> (pop x)
A
> x
(B)
> y
(A B)
```

Структура ячеек после выполнения приведенных выражений показана на рис. 3.9.

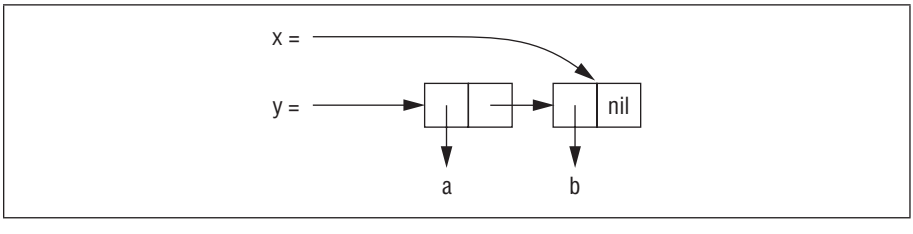

Рис. 3.9. *Эффект от push и pop*

С помощью push можно также определить итеративный вариант функции reverse для списков:

```
(defun our-reverse (lst)
  (let ((acc nil))
    (dolist (elt lst)
      (push elt acc))
    acc))
```

В этом варианте мы начинаем с пустого списка и последовательно кладем на него, как на стопку, элементы исходного. Последний элемент окажется на вершине стопки, то есть в начале списка.

Макрос pushnew похож на push, однако использует adjoin вместо cons:

```
> (let ((x '(a b)))
    (pushnew 'c x)
    (pushnew 'a x)
    x)
(C A B)
```

Элемент a уже присутствует в списке, поэтому не добавляется.

3.13. Точечные пары

Списки, которые могут быть построены с помощью list, называются *правильными списками* (*proper list*). Правильным списком считается либо nil, либо cons-ячейка, cdr которой – также правильный список. Таким образом, можно определить предикат, который возвращает истину только для правильного списка[1]:

```
(defun proper-list? (x)
  (or (null x)
      (and (consp x)
           (proper-list? (cdr x)))))
```

Оказывается, с помощью cons можно создавать не только правильные списки, но и структуры, содержащие ровно два элемента. При этом car соответствует первому элементу структуры, а cdr – второму.

```
> (setf pair (cons 'a 'b))
(A . B)
```

Поскольку эта cons-ячейка не является правильным списком, при отображении ее car и cdr разделяются точкой. Такие ячейки называются *точечными парами* (рис. 3.10).

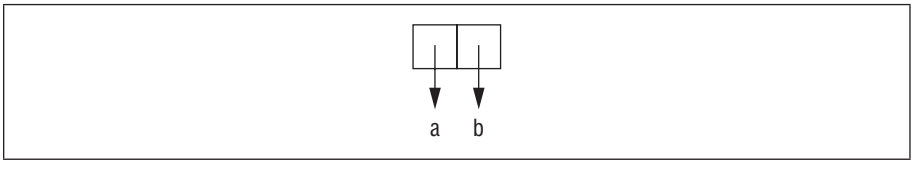

Рис. 3.10. *Ячейка как точечная пара*

Правильные списки можно задавать и в виде набора точечных пар, но они будут отображаться в виде списков:

```
> '(a . (b . (c . nil)))
(A B C)
```

Обратите внимание, как соотносятся ячеечная и точечная нотации – рис. 3.2 и 3.10.

Допустима также смешанная форма записи:

```
> (cons 'a (cons 'b (cons 'c 'd)))
(A B C . D)
```

Структура такого списка показана на рис. 3.11.

---

[1] Вообще говоря, это определение слегка некорректно, т. к. не всегда будет возвращать nil для любого объекта, не являющегося правильным списком. Например, для циклического списка она будет работать бесконечно. Циклические списки обсуждаются в разделе 12.7.

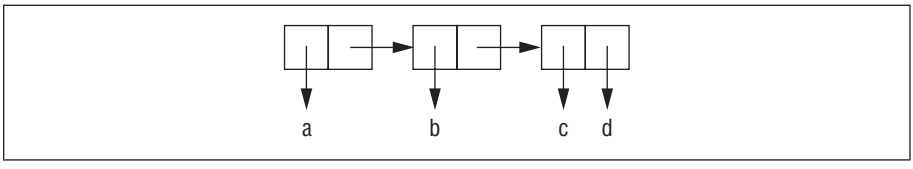

Рис. 3.11. *Список точечных пар*

Таким образом, список (a b) может быть записан аж четырьмя способами:

```
(a . (b . nil))
(a . (b))
(a b . nil)
(a b)
```

и при этом Лисп отобразит их одинаково.

3.14. Ассоциативные списки

Также вполне естественно задействовать cons-ячейки для представления отображений. Список точечных пар называется *ассоциативным списком* (*assoc-list*, *alist*). С помощью него легко определить набор каких-либо правил и соответствий, к примеру:

```
> (setf trans '((+ . "add") (- . "subtract")))
((+ . "add") (- . "subtract"))
```

Ассоциативные списки медленные, но они удобны на начальных этапах работы над программой. В Common Lisp есть встроенная функция assoc для получения по ключу соответствующей ему пары в таком списке:

```
> (assoc '+ trans)
(+ . "add")
> (assoc '* trans)
NIL
```

Если assoc ничего не находит, возвращается nil.

Попробуем определить упрощенный вариант функции assoc:

```
(defun our-assoc (key alist)
  (and (consp alist)
       (let ((pair (car alist)))
         (if (eql key (car pair))
             pair
             (our-assoc key (cdr alist))))))
```

Как и member, реальная функция assoc принимает несколько аргументов по ключу, включая :test и :key. Также Common Lisp определяет assoc-if, которая работает по аналогии с member-if.

3.15. Пример: поиск кратчайшего пути

На рис. 3.12 показана программа, вычисляющая кратчайший путь на графе (или сети). Функции `shortest-path` необходимо сообщить начальную и конечную точки, а также саму сеть, и она вернет кратчайший путь между ними, если он вообще существует.

В этом примере узлам соответствуют символы, а сама сеть представлена как ассоциативный список элементов вида (*узел . соседи*).

Небольшая сеть, показанная на рис. 3.13, может быть записана так:

```
(setf min '((a b c) (b c) (c d)))
```

Найти узлы, в которые можно попасть из узла `a`, поможет функция `assoc`:

```
> (cdr (assoc 'a min))
(B C)
```

```
(defun shortest-path (start end net)
  (bfs end (list (list start)) net))

(defun bfs (end queue net)
  (if (null queue)
      nil
      (let ((path (car queue)))
        (let ((node (car path)))
          (if (eql node end)
              (reverse path)
              (bfs end
                   (append (cdr queue)
                           (new-paths path node net))
                   net))))))

(defun new-paths (path node net)
  (mapcar #'(lambda (n)
              (cons n path))
          (cdr (assoc node net))))
```

Рис. 3.12. *Поиск в ширину*

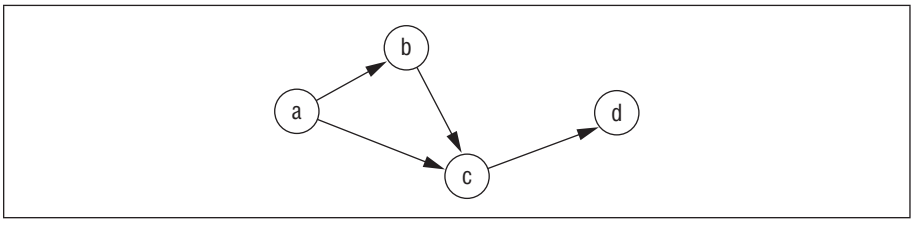

Рис. 3.13. *Простейшая сеть*

Программа на рис. 3.12 реализует *поиск в ширину* (*breadth-first search*). Каждый слой сети исследуется поочередно друг за другом, пока не будет найден нужный элемент или достигнут конец сети. Последовательность исследуемых узлов представляется в виде очереди.

Приведенный на рис. 3.12 код слегка усложняет эту идею, позволяя не только прийти к пункту назначения, но еще и сохранить запись о том, как мы туда добрались. Таким образом, мы оперируем не с очередью узлов, а с очередью пройденных путей.

Поиск выполняется функцией `bfs`. Первоначально в очереди только один элемент – путь к начальному узлу. Таким образом, `shortest-path` вызывает `bfs` с `(list (list start))` в качестве исходной очереди.

Первое, что должна сделать `bfs`, – проверить, остались ли еще непройденные узлы. Если очередь пуста, `bfs` возвращает `nil`, сигнализируя, что путь не был найден. Если же еще есть непроверенные узлы, `bfs` берет первый из очереди. Если `car` этого узла содержит искомый элемент, значит, мы нашли путь до него, и мы возвращаем его, предварительно развернув. В противном случае мы добавляем все дочерние узлы в конец очереди. Затем мы рекурсивно вызываем `bfs` и переходим к следующему слою.

Так как `bfs` осуществляет поиск в ширину, то первый найденный путь будет одновременно самым коротким или одним из кратчайших, если есть и другие пути такой же длины:

```
> (shortest-path 'a 'd min)
(A C D)
```

А вот как выглядит соответствующая очередь во время каждого из вызовов `bfs`:

```
((A))
((B A) (C A))
((C A) (C B A))
((C B A) (D C A))
((D C A) (D C B A))
```

В каждой следующей очереди второй элемент предыдущей очереди становится первым, а первый элемент становится хвостом (`cdr`) любых новых элементов в конце следующей очереди.

Разумеется, приведенный на рис. 3.12 код далек от полноценной эффективной реализации. Однако он убедительно показывает гибкость и удобство списков. В этой короткой программе мы используем списки тремя различными способами: список символов представляет путь, список путей представляет очередь[1] при поиске по ширине, а ассоциативный список изображает саму сеть целиком.

---

[1] В разделе 12.3 будет показано, как реализовать очереди более эффективным образом.

3.16. Мусор

Операции со списками могут быть довольно медленными по ряду причин. Доступ к определенному элементу списка осуществляется не непосредственно по его номеру, а путем последовательного перебора всех предшествующих ему элементов, что может быть существенно медленнее, особенно для больших списков. Так как список является набором вложенных ячеек, то для доступа к элементу приходится выполнить несколько переходов по указателям, в то время как для доступа к элементу массива достаточно просто увеличить позицию указателя на заданное число. Впрочем, намного более затратным может быть выделение новых ячеек.

Автоматическое управление памятью – одна из наиболее ценных особенностей Лиспа. Лисп-система работает с сегментом памяти, называемым *куча* (*heap*). Система владеет информацией об использованной и неиспользованной памяти и выделяет последнюю для размещения новых объектов. Например, функция `cons` выделяет память под создаваемую ею ячейку, поэтому создание новых объектов часто называют *consing*.

Если память будет выделяться, но не освобождаться, то рано или поздно свободная память закончится, и Лисп прекратит работу. Поэтому необходим механизм поиска и освобождения участков кучи, которые более не содержат нужных данных. Память, которая становится ненужной, называется *мусором* и подлежит уборке. Этот процесс называется *сборкой мусора* (*garbge collection*, *GC*).

Как появляется мусор? Давайте немного намусорим:

```
> (setf lst (list 'a 'b 'c))
(A B C)
> (setf lst nil)
NIL
```

Первоначально мы вызвали `list`, которая трижды вызывала `cons`, а она, в свою очередь, выделила новые `cons`-ячейки в куче. После этого мы сделали `lst` пустым списком. Теперь ни одна из трех ячеек, созданных `cons`, не может быть использована[1].

Объекты, которые никаким образом не могут быть доступны для использования, и считаются мусором. Теперь система может безбоязненно повторно использовать память, выделенную `cons`, по своему усмотрению.

Такой механизм управления памятью – огромный плюс для программиста, ведь теперь он не должен заботиться о явном выделении и освобождении памяти. Соответственно исчезают баги, связанные с некор-

[1] Это не всегда так. В toplevel доступны глобальные переменные `*`, `**`, `***`, которым автоматически присваиваются последние три вычисленных значения. До тех пор пока значение связано с одной из вышеупомянутых переменных, мусором оно не считается.

ректным управлением памятью. Утечки памяти и висячие указатели просто невозможны в Лиспе.

Разумеется, за удобство надо платить. Автоматическое управление памятью работает во вред неаккуратному программисту. Затраты на работу с кучей и уборку мусора иногда списывают на consing. Это имеет под собой основания, поскольку, за исключением случаев, когда программа ничего не выбрасывает, большинство выделенных `cons`-ячеек в конце концов окажутся мусором. И беда в том, что работа с consing может быть довольно затратной по сравнению с обычными операциями в программе. Конечно, прогресс не стоит на месте, автоматическое управление памятью становится более эффективным и алгоритмы по сборке мусора совершенствуются, но consing всегда будет иметь определенную стоимость, довольно существенную в некоторых реализациях языка.

Будучи неаккуратным, легко написать программу, выделяющую чрезмерно много памяти. Например, `remove` копирует ячейки, создавая новый список, чтобы не вызвать побочный эффект. Тем не менее такого копирования можно избежать, используя *деструктивные* операции, которые модифицируют существующие данные, а не создают новые. Деструктивные функции подробно рассмотрены в разделе 12.4.

Тем не менее возможно написание программ, которые вообще не выделяют память в процессе выполнения. Часто программа сначала пишется в чисто функциональном стиле с использованием большого количества списков, а по мере развития копирующие функции заменяются на деструктивные в критических к производительности участках кода. Но в этом случае сложно давать конкретные советы, потому что некоторые современные реализации Лиспа управляют памятью настолько хорошо, что иногда выделение новой памяти может быть эффективнее, чем использование уже существующей. Более детально этот вопрос рассматривается в разделе 13.4.

Во всяком случае, consing подходит для прототипов и экспериментов. А если вы воспользуетесь гибкостью, которую дают списки, при изначальном написании программы, то у вас больше шансов, что она доживет до более поздних этапов, на которых понадобится оптимизация.

Итоги главы

1. `Cons`-ячейка – это структура, состоящая из двух объектов. Списки состоят из связанных между собой ячеек.
2. Предикат `equal` менее строг, чем `eql`. Фактически он возвращает истину, если объекты печатаются одинаково.
3. Все объекты в Лиспе ведут себя как указатели. Вам никогда не придется управлять указателями явно.
4. Скопировать список можно с помощью `copy-list`, объединить два списка – с помощью `append`.

5. Кодирование повторов – простой алгоритм сжатия списков, легко реализуемый в Лиспе.
6. Common Lisp имеет богатый набор средств для доступа к элементам списков, и эти функции определены в терминах `car` и `cdr`.
7. Отображающие функции применяют определенную функцию последовательно к каждому элементу или каждому хвосту списка.
8. Операции с вложенными списками сродни операциям с бинарными деревьями.
9. Чтобы оценить корректность рекурсивной функции, достаточно убедиться, что она соответствует нескольким требованиям.
10. Списки могут рассматриваться как множества. Для работы с множествами в Лиспе есть ряд встроенных функций.
11. Аргументы по ключу не являются обязательными и определяются не положением, а по соответствующей метке.
12. Список – подтип последовательности. Common Lisp имеет множество функций для работы с последовательностями.
13. `Cons`-ячейка, не являющаяся правильным списком, называется точечной парой.
14. С помощью списков точечных пар можно представить элементы отображения. Такие списки называются ассоциативными.
15. Автоматическое управление памятью освобождает программиста от ручного выделения памяти, но большое количество мусора может замедлить работу программы.

Упражнения

1. Представьте следующие списки в виде ячеек:

   (a) `(a b (c d))`
   (b) `(a (b (c (d))))`
   (c) `(((a b) c) d)`
   (d) `(a (b . c) . d)`

2. Напишите свой вариант функции `union`, который сохраняет порядок следования элементов согласно исходным спискам:

```
> (new-union '(a b c) '(b a d))
(A B C D)
```

3. Напишите функцию, определяющую количество повторений (с точки зрения `eql`) каждого элемента в заданном списке и сортирующую их по убыванию встречаемости:

```
> (occurrences '(a b a d a c d c a))
((A . 4) (C . 2) (D . 2) (B . 1))
```

4. Почему `(member '(a) '((a) (b)))` возвращает `nil`?

5. Функция pos+ принимает список и возвращает новый, каждый элемент которого увеличен по сравнению с исходным на его положение в списке:

```
> (pos+ '(7 5 1 4))
(7 6 3 7)
```

   Определите эту функцию с помощью: (a) рекурсии, (b) итерации и (c) mapcar.

6. После долгих лет раздумий государственная комиссия приняла постановление, согласно которому cdr указывает на первый элемент списка, а car – на его остаток. Определите следующие функции, удовлетворяющие этому постановлению:

   (a) cons
   (b) list[1]
   (c) length (для списков)
   (d) member (для списков, без ключевых параметров)

7. Измените программу на рис. 3.6 таким образом, чтобы она создавала меньшее количество ячеек. (Подсказка: используйте точечные пары.)

8. Определите функцию, печатающую заданный список в точечной нотации:

```
> (showdots '(a b c))
(A . (B . (C . NIL)))
NIL
```

9. Напишите программу, которая ищет *наиболее длинный* путь в сети, не содержащий повторений (раздел 3.15). Сеть может содержать циклы.

[1] Задачу 6b пока что не получится решить с помощью имеющихся у вас знаний. Вам потребуется остаточный аргумент (&rest), который вводится на стр. 114. На эту оплошность указал Рикардо Феррейро де Оливьера. – *Прим. перев.*

4

Специализированные структуры данных

В предыдущей главе были рассмотрены списки – наиболее универсальные структуры для хранения данных. В этой главе будут рассмотрены другие способы хранения данных в Лиспе: массивы (а также векторы и строки), структуры и хеш-таблицы. Они не настолько гибки, как списки, но позволяют осуществлять более быстрый доступ и занимают меньше места.

В Common Lisp есть еще один тип структур – *экземпляры объектов* (*instance*). О них подробно рассказано в главе 11, описывающей CLOS.

4.1. Массивы

В Common Lisp массивы создаются с помощью функции `make-array`, первым аргументом которой выступает список размерностей. Создадим массив 2×3:

```
> (setf arr (make-array '(2 3) :initial-element nil))
#<Simple-Array T (2 3) BFC4FE>
```

Многомерные массивы в Common Lisp могут иметь по меньшей мере 7 размерностей, а в каждом измерении поддерживается хранение не менее 1023 элементов[1].

Аргумент `:initial-element` не является обязательным. Если он используется, то устанавливает начальное значение каждого элемента массива. Поведение системы при попытке получить значение элемента массива, не инициализированного начальным значением, не определено.

---

[1] В конкретной реализации ограничение сверху на количество размерностей и элементов может быть и больше. – *Прим. перев.*

Чтобы получить элемент массива, воспользуйтесь aref. Как и большинство других функций доступа в Common Lisp, aref начинает отсчет элементов с нуля:

```
> (aref arr 0 0)
NIL
```

Новое значение элемента массива можно установить, используя setf вместе с aref:

```
> (setf (aref arr 0 0) 'b)
B
> (aref arr 0 0)
B
```

Как и списки, массивы могут быть заданы буквально с помощью синтаксиса #*n*a, где *n* – количество размерностей массива. Например, текущее состояние массива arr может быть задано так:

```
#2a((b nil nil) (nil nil nil))
```

Если глобальная переменная *print-array*[1] установлена в t, массивы будут печататься в таком виде:

```
> (setf *print-array* t)
T
> arr
#2A((B NIL NIL) (NIL NIL NIL))
```

Для создания одномерного массива можно вместо списка размерностей в качестве первого аргумента передать функции make-array целое число:

```
> (setf vec (make-array 4 :initial-element nil))
#(NIL NIL NIL NIL)
```

Одномерный массив также называют *вектором*. Создать и заполнить вектор можно с помощью функции vector:

```
> (vector "a" 'b 3)
#("a" B 3)
```

Как и массив, который может быть задан буквально с помощью синтаксиса #*n*a, вектор может быть задан буквально с помощью синтаксиса #().

Хотя доступ к элементам вектора может осуществить aref, для работы с векторами есть более быстрая функция svref:

```
> (svref vec 0)
NIL
```

[1] В вашей реализации *print-array* может изначально иметь значение t. – *Прим. перев.*

Префикс «sv» расшифровывается как «simple vector». По умолчанию все векторы создаются как простые векторы.[1]

4.2. Пример: бинарный поиск

В этом разделе в качестве примера показано, как написать функцию поиска элемента в отсортированном векторе. Если нам известно, что элементы вектора расположены в определенном порядке, то поиск нужного элемента может быть выполнен быстрее, чем с помощью функции `find` (стр. 80). Вместо того чтобы последовательно проверять элемент за элементом, мы сразу перемещаемся в середину вектора. Если средний элемент соответствует искомому, то поиск закончен. В противном случае мы продолжаем поиск в правой или левой половине в зависимости от того, больше или меньше искомого значения этот средний элемент вектора.

На рис. 4.1 приведена программа, которая работает подобным образом. Она состоит из двух функций: `bin-search`[2] определяет границы поиска и передает управление функции `finder`, которая ищет соответствующий элемент между позициями `start` и `end` вектора `vec`.

```
(defun bin-search (obj vec)
  (let ((len (length vec)))
    (and (not (zerop len))
         (finder obj vec 0 (- len 1)))))

(defun finder (obj vec start end)
  (let ((range (- end start)))
    (if (zerop range)
        (if (eql obj (aref vec start))
            obj
            nil)
        (let ((mid (+ start (round (/ range 2)))))
          (let ((obj2 (aref vec mid)))
            (if (< obj obj2)
                (finder obj vec start (- mid 1))
                (if (> obj obj2)
                    (finder obj vec (+ mid 1) end)
                    obj)))))))
```

Рис. 4.1. *Поиск в отсортированном векторе*

---

[1] Простой массив не является ни расширяемым (adjustable), ни предразмещенным (displaced) и не имеет указатель заполнения (fill-pointer). По умолчанию все массивы простые. Простой вектор – это одномерный простой массив, который может содержать элементы любого типа.

[2] Приведенная версия `bin-search` не работает, если ей передать объект, меньший наименьшего из элементов вектора. Оплошность найдена Ричардом Грином. – *Прим. перев.*

Когда область поиска сокращается до одного элемента, возвращается сам элемент в случае его соответствия искомому значению obj, в противном случае – nil. Если область поиска состоит из нескольких элементов, определяется ее средний элемент – obj2 (функция round возвращает ближайшее целое число), который сравнивается с искомым элементом obj. Если obj меньше obj2, поиск продолжается рекурсивно в левой половине вектора, в противном случае – в правой половине. Остается вариант obj = obj2, но это значит, что искомый элемент найден и мы просто его возвращаем.

Если вставить следующую строку в начало определения функции finder,

```
(format t "~A~%" (subseq vec start (+ end 1)))
```

мы сможем наблюдать за процессом отсечения половин на каждом шаге:

```
> (bin-search 3 #(0 1 2 3 4 5 6 7 8 9))
#(0 1 2 3 4 5 6 7 8 9)
#(0 1 2 3)
#(3)
3
```

Договоренности о комментировании

В Common Lisp все, что следует за точкой с запятой, считается комментарием. Многие программисты используют последовательно несколько знаков комментирования, разделяя комментарии по уровням: четыре точки с запятой в заглавии файла, три – в описании функции или макроса, две – для пояснения последующей строки, одну – в той же строке, что и поясняемый код. Таким образом, с использованием общепринятых норм комментирования начало кода на рис. 4.1 будет выглядеть так:

```
;;;; Инструменты для операций с отсортированными векторами

;;; Находит элемент в отсортированном векторе

(defun bin-search (obj vec)
  (let ((len (length vec)))
    ;; если это действительно вектор, применяем к нему finder
    (and (not (zerop len)) ; возвращает nil, если вектор пуст
         (finder obj vec 0 (- len 1)))))
```

Многострочные комментарии удобно делать с помощью макроса чтения #|...|#. Все, что находится между #| и |#, игнорируется считывателем.°

4.3. Строки и знаки[1]

Строки – это векторы, состоящие из знаков. Строкой принято называть набор знаков, заключенный в двойные кавычки. Одиночный знак, например c, задается так: #\c.

Каждый знак соответствует определенному целому числу, как правило, (хотя и не обязательно) в соответствии с ASCII. В большинстве реализаций есть функция char-code, которая возвращает связанное со знаком число, и функция code-char, выполняющая обратное преобразование.°

Для сравнения знаков используются следующие функции: char< (меньше), char<= (меньше или равно), char= (равно), char>= (больше или равно), char> (больше) и char/= (не равно). Они работают так же, как и функции сравнения чисел, которые рассматриваются на стр. 157.

```
> (sort "elbow" #'char<)
"below"
```

Поскольку строки – это массивы, то к ним применимы все операции с массивами. Например, получить знак, находящийся в конкретной позиции, можно с помощью aref:

```
> (aref "abc" 1)
#\b
```

Однако эта операция может быть выполнена быстрее с помощью специализированной функции char:

```
> (char "abc" 1)
#\b
```

Функция char, как и aref, может быть использована вместе с setf для замены элементов:

```
> (let ((str (copy-seq "Merlin")))
    (setf (char str 3) #\k)
    str)
"Merkin"
```

Чтобы сравнить две строки, можно воспользоваться известной вам функцией equal, но есть также и специализированная string-equal, которая к тому же не учитывает регистр букв:

```
> (equal "fred" "fred")
T
> (equal "fred" "Fred")
NIL
```

---

[1] Здесь и далее во избежание путаницы между символами в терминах Лиспа и символами-знаками, из которых состоят строки (буквами, цифрами и другими типами символов), мы будем называть последние просто «знаками». В случае когда путаница исключена, может использоваться также более привычный термин «символ». – *Прим. перев.*

```
> (string-equal "fred" "Fred")
T
```

В Common Lisp определен большой набор функций для сравнения и прочих манипуляций со строками. Они перечислены в приложении D, начиная со стр. 378.

Есть несколько способов создания строк. Самый общий – с помощью функции format. При использовании nil в качестве ее первого аргумента format вернет строку, вместо того чтобы ее напечатать:

```
> (format nil "~A or ~A" "truth" "dare")
"truth or dare"
```

Но если вам нужно просто соединить несколько строк, можно воспользоваться concatenate, которая принимает тип результата и одну или несколько последовательностей:

```
> (concatenate 'string "not " "to worry")
"not to worry"
```

4.4. Последовательности

Тип *последовательность* (*sequence*) в Common Lisp включает в себя списки и векторы (а значит, и строки). Многие функции из тех, которые мы ранее использовали для списков, на самом деле определены для любых последовательностей. Это, например, remove, length, subseq, reverse, sort, every, some. Таким образом, функция, определенная нами на стр. 62, будет работать и с другими видами последовательностей:

```
> (mirror? "abba")
T
```

Мы уже знаем некоторые функции для доступа к элементам последовательностей: nth для списков, aref и svref для векторов, char для строк. Доступ к элементу последовательности любого типа может быть осуществлен с помощью elt:

```
> (elt '(a b c) 1)
B
```

Специализированные функции работают быстрее, и использовать elt рекомендуется только тогда, когда тип последовательности заранее не известен.

С помощью elt функция mirror? может быть оптимизирована для векторов:

```
(defun mirror? (s)
  (let ((len (length s)))
    (and (evenp len)
         (do ((forward 0 (+ forward 1))
              (back (- len 1) (- back 1)))
             ((or (> forward back)
```

```
                    (not (eql (elt s forward)
                              (elt s back))))
                (> forward back))))))
```

Эта версия по-прежнему будет работать и со списками, однако она более приспособлена для векторов. Регулярное использование последовательного доступа к элементам списка довольно затратно, а непосредственного доступа к нужному элементу они не предоставляют. Для векторов же стоимость доступа к любому элементу не зависит от его положения.

Многие функции, работающие с последовательностями, имеют несколько аргументов по ключу:

Параметр	Назначение	По умолчанию
:key	Функция, применяемая к каждому элементу	identity
:test	Предикат для сравнения	eql
:from-end	Если t, работа с конца	nil
:start	Индекс элемента, с которого начинается выполнение	0
:end	Если задан, то индекс элемента, на котором следует остановиться	nil

Одна из функций, которая принимает все эти аргументы, – position. Она возвращает положение определенного элемента в последовательности или nil в случае его отсутствия. Посмотрим на роль аргументов по ключу на примере position:

```
> (position #\a "fantasia")
1
> (position #\a "fantasia" :start 3 :end 5)
4
```

Во втором случае поиск выполняется между четвертым и шестым элементом. Аргумент :start ограничивает подпоследовательность слева, :end ограничивает справа или же не ограничивает вовсе, если этот аргумент не задан.

Задавая параметр :from-end:

```
> (position #\a "fantasia" :from-end t)
7
```

мы получаем позицию элемента, ближайшего к концу последовательности. Но позиция элемента вычисляется как обычно, то есть от начала списка (однако поиск элемента производится с конца списка).

Параметр :key определяет функцию, применяемую к каждому элементу перед сравнением его с искомым:

```
> (position 'a '((c d) (a b)) :key #'car)
1
```

В этом примере мы поинтересовались, car какого элемента содержит a.

Параметр :test определяет, с помощью какой функции будут сравниваться элементы. По умолчанию используется eql. Если вам необходимо сравнивать списки, придется воспользоваться функцией equal:

```
> (position '(a b) '((a b) (c d)))
NIL
> (position '(a b) '((a b) (c d)) :test #'equal)
0
```

Аргумент :test может быть любой функцией от двух элементов. Например, с помощью < можно найти первый элемент, больший заданного:

```
> (position 3 '(1 0 7 5) :test #'<)
2
```

С помощью subseq и position можно разделить последовательность на части. Например, функция

```
(defun second-word (str)
  (let ((p1 (+ (position #\  str) 1)))
    (subseq str p1 (position #\  str :start p1))))
```

возвращает второе слово в предложении:

```
> (second-word "Form follows function.")
"follows"
```

Поиск элементов, удовлетворяющих заданному предикату, осуществляется с помощью position-if. Она принимает функцию и последовательность, возвращая положение первого встреченного элемента, удовлетворяющего предикату:

```
> (position-if #'oddp '(2 3 4 5))
1
```

Эта функция принимает все вышеперечисленные аргументы по ключу, за исключением :test.

Также для последовательностей определены функции, аналогичные member и member-if. Это find (принимает все аргументы по ключу) и find-if (принимает все аргументы, кроме :test):

```
> (find #\a "cat")
#\a
> (find-if #'characterp "ham")
#\h
```

В отличие от member и member-if, они возвращают только сам найденный элемент.

Вместо find-if иногда лучше использовать find с ключом :key. Например, выражение:

```
(find-if #'(lambda (x)
             (eql (car x) 'complete))
         lst)
```

будет выглядеть понятнее в виде:

```
(find 'complete lst :key #'car)
```

Функции remove (стр. 39) и remove-if работают с последовательностями любого типа. Разница между ними точно такая же, как между find и find-if. Связанная с ними функция remove-duplicates удаляет все повторяющиеся элементы последовательности, кроме последнего:

```
> (remove-duplicates "abracadabra")
"cdbra"
```

Эта функция использует все аргументы по ключу, рассмотренные в таблице выше.

Функция reduce сводит последовательность в одно значение. Она принимает функцию, по крайней мере, с двумя аргументами и последовательность. Заданная функция первоначально применяется к первым двум элементам последовательности, а затем последовательно к полученному результату и следующему элементу последовательности. Последнее полученное значение будет возвращено как результат reduce. Таким образом, вызов:

```
(reduce #'fn '(a b c d))
```

будет эквивалентен

```
(fn (fn (fn 'a 'b) 'c) 'd)
```

Хорошее применение reduce – расширение набора аргументов для функций, которые принимают только два аргумента. Например, чтобы получить пересечение трех или более списков, можно написать:

```
> (reduce #'intersection '((b r a d 's) (b a d) (c a t)))
(A)
```

4.5. Пример: разбор дат

В качестве примера операций с последовательностями в этом разделе приводится программа для разбора дат. Мы напишем программу, которая превращает строку типа "16 Aug 1980" в целые числа, соответствующие дню, месяцу и году.

Программа на рис. 4.2 содержит некоторые функции, которые потребуются нам в дальнейшем. Первая, tokens, выделяет знаки из строки. Функция tokens принимает строку и предикат, возвращая список подстрок, все знаки в которых удовлетворяют этому предикату. Приведем пример. Пусть используется функция alpha-char-p – предикат, справедливый для буквенных знаков. Тогда получим:

```
> (tokens "ab12 3cde.f" #'alpha-char-p 0)
("ab" "cde" "f")
```

Все остальные знаки, не удовлетворяющие данному предикату, рассматриваются как пробельные.

```
(defun tokens (str test start)
 (let ((p1 (position-if test str :start start)))
   (if p1
       (let ((p2 (position-if #'(lambda (c)
                                  (not (funcall test c)))
                              str :start p1)))
         (cons (subseq str p1 p2)
               (if p2
                   (tokens str test p2)
                   nil)))
       nil)))

(defun constituent (c)
  (and (graphic-char-p c)
       (not (char= c #\ ))))
```

Рис. 4.2. *Распознавание символов*

Функция `constituent` будет использоваться в качестве предиката для `tokens`. В Common Lisp к *печатным знакам* (*graphic characters*) относятся все знаки, которые видны при печати, а также пробел. Вызов `tokens` с функцией `constituent` будет выделять подстроки, состоящие из печатных знаков:

```
> (tokens "ab12 3cde.f
          gh" #'constituent 0)
("ab12" "3cde.f" "gh")
```

На рис. 4.3 показаны функции, выполняющие разбор дат.

```
(defun parse-date (str)
  (let ((toks (tokens str #'constituent 0)))
    (list (parse-integer (first toks))
          (parse-month (second toks))
          (parse-integer (third toks)))))

(defconstant month-names
  #("jan" "feb" "mar" "apr" "may" "jun"
    "jul" "aug" "sep" "oct" "nov" "dec"))

(defun parse-month (str)
  (let ((p (position str month-names
                     :test #'string-equal)))
    (if p
        (+ p 1)
        nil)))
```

Рис. 4.3. *Функции для разбора дат*

Функция parse-date принимает дату, записанную в указанной форме, и возвращает список целых чисел, соответствующих ее компонентам:

```
> (parse-date "16 Aug 1980")
(16 8 1980)
```

Эта функция делит строку на части и применяет parse-month и parse-integer к полученным частям. Функция parse-month не чувствительна к регистру, так как сравнивает строки с помощью string-equal. Для преобразования строки, содержащей число, в само число, используется встроенная функция parse-integer.

Однако если бы такой функции в Common Lisp изначально не было, нам пришлось бы определить ее самостоятельно:

```
(defun read-integer (str)
  (if (every #'digit-char-p str)
      (let ((accum 0))
        (dotimes (pos (length str))
          (setf accum (+ (* accum 10)
                         (digit-char-p (char str pos)))))
        accum)
      nil))
```

Определенная нами функция read-integer показывает, как в Common Lisp преобразовать набор знаков в число. Она использует особенность функции digit-char-p, которая проверяет, является ли аргумент цифрой, и возвращает саму цифру, если это так.

4.6. Структуры

Структура может рассматриваться как более продвинутый вариант вектора. Предположим, что нам нужно написать программу, отслеживающую положение набора параллелепипедов. Каждое такое тело можно представить в виде вектора, состоящего из трех элементов: высота, ширина и глубина. Программу будет проще читать, если вместо простых svref мы будем использовать специальные функции:

```
(defun block-height (b) (svref b 0))
```

и так далее. Можете считать структуру таким вектором, у которого все эти функции уже заданы.

Определить структуру можно с помощью defstruct. В простейшем случае достаточно задать имена структуры и ее полей:

```
(defstruct point
  x
  y)
```

Мы определили структуру point, имеющею два поля, x и y. Кроме того, неявно были заданы функции: make-point, point-p, copy-point, point-x, point-y.

В разделе 2.3 мы упоминали о способности Лисп-программ писать другие Лисп-программы. Это один из наглядных примеров: при вызове defstruct самостоятельно определяет все необходимые функции. Научившись работать с макросами, вы сами сможете делать похожие вещи. (Вы даже смогли бы написать свою версию defstruct, если бы в этом была необходимость.)

Каждый вызов make-point возвращает вновь созданный экземпляр структуры point. Значения полей могут быть изначально заданы с помощью соответствующих аргументов по ключу:

```
> (setf p (make-point :x 0 :y 0))
#S(POINT X 0 Y 0)
```

Функции доступа к полям структуры определены не только для чтения полей, но и для задания значений с помощью setf:

```
> (point-x p)
0
> (setf (point-y p) 2)
2
> p
#S(POINT X 0 Y 2)
```

Определение структуры также приводит к определению одноименного типа. Каждый экземпляр point принадлежит типу point, затем structure, затем atom и t. Таким образом, использование point-p равносильно проверке типа:

```
> (point-p p)
T
> (typep p 'point)
T
```

Функция typep проверяет объект на принадлежность к заданному типу.

Также можно задать значения полей по умолчанию, если заключить имя соответствующего поля в список и поместить в него выражение для вычисления этого значения.

```
(defstruct polemic
  (type (progn
          (format t "What kind of polemic was it? ")
          (read)))
  (effect nil))
```

Вызов make-polemic без дополнительных аргументов установит исходные значения полей:

```
> (make-polemic)
What kind of polemic was it? scathing
#S(POLEMIC TYPE SCATHING EFFECT NIL)
```

Кроме того, можно управлять такими вещами, как способ отображения структуры и префикс имен функций для доступа к полям. Вот более развитый вариант определения структуры point:

```
(defstruct (point (:conc-name p)
                  (:print-function print-point))
  (x 0)
  (y 0))

(defun print-point (p stream depth)
  (format stream "#<~A, ~A>" (px p) (py p)))
```

Аргумент :conc-name задает префикс, с которого будут начинаться имена функций для доступа к полям структуры. По умолчанию он равен point-, а в новом определении это просто p. Отход от варианта по умолчанию делает код менее читаемым, поэтому использовать более короткий префикс стоит, только если вам предстоит постоянно пользоваться функциями доступа к полям.

Параметр :print-function – это *имя* функции, которая будет вызываться для печати объекта, когда его нужно будет отобразить (например, в top-level). Такая функция должна принимать три аргумента: сам объект; поток, куда он будет напечатан; третий аргумент обычно не требуется и может быть проигнорирован[1]. С потоками ввода-вывода мы познакомимся подробнее в разделе 7.1. Сейчас достаточно сказать, что второй аргумент, поток, может быть передан функции format.

Функция print-point будет отображать структуру в такой сокращенной форме:

```
> (make-point)
#<0,0>
```

4.7. Пример: двоичные деревья поиска

Поскольку в Common Lisp имеется встроенная функция sort, вам, скорее всего, не придется самостоятельно писать процедуры поиска. В этом разделе показано, как решить похожую задачу, для которой нет встроенной функции: поддержание набора объектов в отсортированном виде. Мы рассмотрим метод хранения объектов в *двоичном дереве поиска* (*BST*). Сбалансированное BST позволяет искать, добавлять или удалять элементы за время, пропорциональное $\log n$, где n – количество объектов в наборе.

BST – это бинарное дерево, в котором для каждого элемента и некоторой функции упорядочения (пусть это будет функция <) соблюдается правило: левый дочерний элемент < элемента-родителя, и сам элемент >

[1] В ANSI Common Lisp вы можете передавать в :print-object функцию двух параметров. Кроме того, существует макрос print-unreadable-object, который может быть использован для отображения объекта в виде #<...>.

правого дочернего элемента. На рис. 4.4 показан пример BST, упорядоченного с помощью функции <.

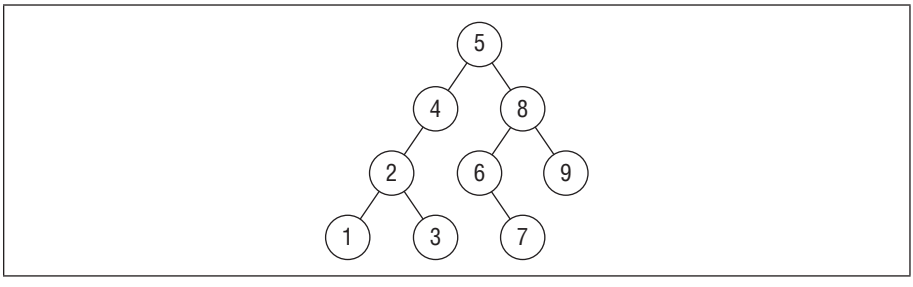

Рис. 4.4. *Двоичное дерево поиска*

Программа на рис. 4.5 содержит утилиты для вставки и поиска объектов в BST. В качестве основной структуры данных используются узлы. Каждый узел имеет три поля: в одном хранится сам объект, в двух других – левый и правый потомки. Можно рассматривать узел как cons-ячейку с одним car и двумя cdr.

BST может быть либо nil, либо узлом, поддеревья которого (l и r) также являются BST. Продолжим дальнейшую аналогию со списками. Как список может быть создан последовательностью вызовов cons, так и бинарное дерево может быть построено с помощью вызовов bst-insert. Этой функции необходимо сообщить объект, дерево и функцию упорядочения.

```
> (setf nums nil)
 NIL
> (dolist (x '(5 8 4 2 1 9 6 7 3))
    (setf nums (bst-insert x nums #'<)))
NIL
```

Теперь дерево nums соответствует рис. 4.4.

Функция bst-find, которая ищет объекты в дереве, принимает те же аргументы, что и bst-insert. Аналогия со списками станет еще понятнее, если мы сравним определения bst-find и our-member (стр. 33).

Как и member, bst-find возвращает не сам элемент, а его поддерево:

```
> (bst-find 12 nums #'<)
NIL
> (bst-find 4 nums #'<)
#<4>
```

Такое представление позволяет нам различать случаи, в которых искомый элемент не найден (nil) и в которых успешно найден элемент nil.

```
(defstruct (node (:print-function
                  (lambda (n s d)
                    (format s "#<~A>" (node-elt n)))))
  elt (l nil) (r nil))

(defun bst-insert (obj bst <)
  (if (null bst)
      (make-node :elt obj)
      (let ((elt (node-elt bst)))
        (if (eql obj elt)
            bst
            (if (funcall < obj elt)
                (make-node
                  :elt elt
                  :l   (bst-insert obj (node-l bst) <)
                  :r   (node-r bst))
                (make-node
                  :elt elt
                  :r   (bst-insert obj (node-r bst) <)
                  :l   (node-l bst)))))))

(defun bst-find (obj bst <)
  (if (null bst)
      nil
      (let ((elt (node-elt bst)))
        (if (eql obj elt)
            bst
            (if (funcall < obj elt)
                (bst-find obj (node-l bst) <)
                (bst-find obj (node-r bst) <))))))

(defun bst-min (bst)
  (and bst
       (or (bst-min (node-l bst)) bst)))

(defun bst-max (bst)
  (and bst
       (or (bst-max (node-r bst)) bst)))
```

Рис. 4.5. *Двоичные деревья поиска: поиск и вставка*

Нахождение наибольшего и наименьшего элементов BST также не составляет особого труда. Чтобы найти минимальный элемент, мы идем по дереву, всегда выбирая левую ветвь (`bst-min`). Аналогично, следуя правым поддеревьям, мы получим наибольший элемент (`bst-max`):

```
> (bst-min nums)
#<1>
> (bst-max nums)
#<9>
```

Удаление элемента из бинарного дерева выполняется так же быстро, но соответствующий код (рис. 4.6) выглядит сложнее.

```
(defun bst-remove (obj bst <)
  (if (null bst)
      nil
      (let ((elt (node-elt bst)))
        (if (eql obj elt)
            (percolate bst)
            (if (funcall < obj elt)
                (make-node
                  :elt elt
                  :l (bst-remove obj (node-l bst) <)
                  :r (node-r bst))
                (make-node
                  :elt elt
                  :r (bst-remove obj (node-r bst) <)
                  :l (node-l bst)))))))

(defun percolate (bst)
  (let ((l (node-l bst)) (r (node-r bst)))
    (cond ((null l) r)
          ((null r) l)
          (t (if (zerop (random 2))
                 (make-node :elt (node-elt (bst-max l))
                            :r r
                            :l (bst-remove-max l))
                 (make-node :elt (node-elt (bst-min r))
                            :r (bst-remove-min r)
                            :l l))))))

(defun bst-remove-min (bst)
  (if (null (node-l bst))
      (node-r bst)
      (make-node :elt (node-elt bst)
                 :l   (bst-remove-min (node-l bst))
                 :r   (node-r bst))))

(defun bst-remove-max (bst)
  (if (null (node-r bst))
      (node-l bst)
      (make-node :elt (node-elt bst)
                 :l   (node-l bst)
                 :r   (bst-remove-max (node-r bst)))))
```

Рис. 4.6. *Двоичные деревья поиска: удаление*

Функция bst-remove[1] принимает объект, дерево и функцию упорядочения и возвращает это же дерево без заданного элемента. Как и remove, bst-remove не модифицирует исходное дерево:

```
> (setf nums (bst-remove 2 nums #'<))
#<5>
> (bst-find 2 nums #'<)
NIL
```

Теперь дерево nums соответствует рис. 4.7. (Другой возможный случай – подстановка элемента 1 на место 2.)

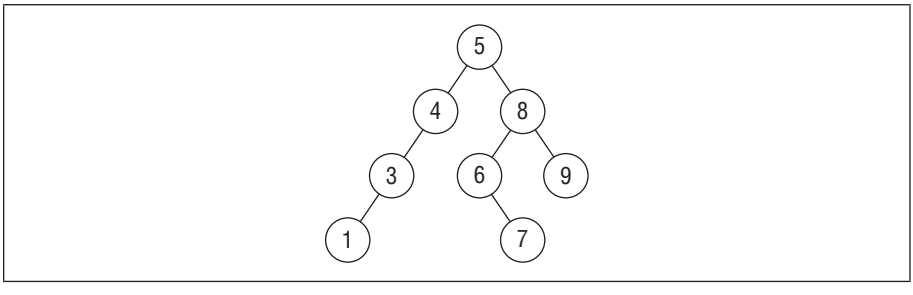

Рис. 4.7. *Двоичное дерево поиска после удаления одного из элементов*

Удаление – более затратная процедура, так как под удаляемым объектом появляется незанятое место, которое должно быть заполнено одним из поддеревьев этого объекта. Этим занимается функция percolate. Она замещает элемент дерева одним из его поддеревьев, затем замещает это поддерево одним из *его* поддеревьев и т. д.

Чтобы сбалансировать дерево, percolate случайным образом выбирает одно из двух поддеревьев. Выражение (random 2) вернет либо 0, либо 1, в результате (zerop (random 2)) будет истинно в половине случаев.

Теперь, когда мы преобразовали набор объектов в бинарное дерево, последовательный обход его элементов даст нам их в порядке возрастания.

---

[1] Версия bst-remove, приведенная в оригинале книги, содержит баг.
Крис Стовер сообщает: «Задания (left child) < node < (right child) для каждого узла недостаточно. Необходимо более строгое ограничение: max(left subtree) < node < min(right subtree). Без него функция bst-traverse, приведенная на рис. 4.8, не обязательно перечислит элементы в порядке возрастания. (Пример: дерево с основанием 1 имеет только правого потомка 2, а он имеет только правого потомка 0.) К счастью, функция bst-insert, представленная на рис. 4.5, корректна. С другой стороны, корректный порядок BST после применения bst-remove не гарантируется. Удаляемый внутренний узел необходимо заменять либо максимальным узлом левого поддерева, либо минимальным узлом правого поддерева, а приведенная функция этого не делает.»
В программе на рис. 4.6 данный баг уже исправлен. – *Прим. перев.*

```
(defun bst-traverse (fn bst)
  (when bst
    (bst-traverse fn (node-l bst))
    (funcall fn (node-elt bst))
    (bst-traverse fn (node-r bst))))
```

Рис. 4.8. *Двоичные деревья поиска: обход*

Для этой цели определена функция `bst-traverse` (рис. 4.8), которая применяет к каждому элементу дерева функцию `fn`.

```
> (bst-traverse #'princ nums)
13456789
NIL
```

(Функция `princ` всего лишь отображает отдельный объект.)

Код, представленный в этом разделе, служит основой для реализации двоичных деревьев поиска. Вероятно, вы захотите как-то усовершенствовать его в соответствии с вашими потребностями. Например, каждый узел в текущей реализации имеет лишь одно поле `elt`, в то время как может оказаться полезным введение двух полей – ключ и значение. Данная версия хотя и не поддерживает такую возможность, но позволяет ее легко реализовать.

Двоичные деревья поиска могут использоваться не только для управления отсортированным набором объектов. Их применение оптимально, когда вставки и удаления узлов имеют равномерное распределение. Это значит, что для работы, например, с очередями, BST *не* лучший выбор. Хотя вставки в очередях вполне могут быть распределены равномерно, удаления будут всегда осуществляться с конца. Это будет приводить к разбалансировке дерева, и вместо ожидаемой оценки $O(\log n)$ мы получим $O(n)$. Кроме того, для моделирования очередей удобнее использовать обычный список просто потому, что BST будет вести себя в конечном счете так же, как и список.°

4.8. Хеш-таблицы

В главе 3 было показано, что списки могут использоваться для представления множеств и отображений. Для достаточно больших массивов данных (начиная уже с 10 элементов) использование хеш-таблиц существенно увеличит производительность. Хеш-таблицу можно создать с помощью функции `make-hash-table`, которая не требует обязательных аргументов:

```
> (setf ht (make-hash-table))
#<Hash-Table BF0A96>
```

Хеш-таблицы, как и функции, при печати отображаются в виде `#<...>`.

Хеш-таблица, как и ассоциативный список, – это способ ассоциирования пар объектов. Чтобы получить значение, связанное с заданным ключом, достаточно вызвать gethash с этим ключом и таблицей. По умолчанию gethash возвращает nil, если не находит искомого элемента.

```
> (gethash 'color ht)
NIL
NIL
```

Здесь мы впервые сталкиваемся с важной особенностью Common Lisp: выражение может возвращать несколько значений. Функция gethash возвращает два. Первое значение ассоциировано с ключом. Второе значение, если оно nil (как в нашем примере), означает, что искомый элемент не был найден. Почему мы не можем судить об этом из первого nil? Дело в том, что элементом, связанным с ключом color, может оказаться nil, и gethash вернет его, но в этом случае в качестве второго значения – t.

Большинство реализаций выводит последовательно все возвращаемые значения, но если результат многозначной функции используется другой функцией, то ей передается лишь первое значение. В разделе 5.5 объясняется, как получать и использовать сразу несколько значений.

Чтобы сопоставить новое значение какому-либо ключу, используем setf вместе с gethash:

```
> (setf (gethash 'color ht) 'red)
RED
```

Теперь gethash вернет вновь установленное значение:

```
> (gethash 'color ht)
RED
T
```

Второе значение подтверждает, что gethash вернул реально имеющийся в таблице объект, а не значение по умолчанию.

Объекты, хранящиеся в хеш-таблицах, могут иметь любой тип. Например, при желании сопоставить каждой функции ее краткое описание можно создать таблицу, в которой ключами будут функции, а значениями – строки:

```
> (setf bugs (make-hash-table))
#<Hash-Table BF4C36>
> (push "Doesn't take keyword arguments. "
        (gethash #'our-member bugs))
("Doesn't take keyword arguments. ")
```

Так как по умолчанию gethash возвращает nil, вызов push эквивалентен setf, и мы просто кладем нашу строку на пустой список. (Определение функции our-member находится на стр. 39.)

Хеш-таблицы также можно использовать вместо списков для представления множеств. Они существенно ускоряют поиск значений и их удаление в случаях больших объемов данных. Чтобы добавить элемент

в множество, представленное в виде хеш-таблицы, используйте setf вместе с gethash:

```
> (setf fruit (make-hash-table))
#<Hash-Table BFDE76>
> (setf (gethash 'apricot fruit) t)
T
```

Проверка на принадлежность элемента множеству выполняется с помощью gethash:

```
> (gethash 'apricot fruit)
T
T
```

По умолчанию gethash возвращает nil, поэтому вновь созданная хеш-таблица представляет собой пустое множество.

Чтобы удалить элемент из множества, можно воспользоваться remhash:

```
> (remhash 'apricot fruit)
T
```

Возвращая t, remhash сигнализирует, что искомый элемент был найден и успешно удален.

Для итерации по хеш-таблице существует maphash, которой необходимо передать функцию двух аргументов и саму таблицу. Эта функция будет вызвана с каждой имеющейся в таблице парой ключ-значение в произвольном порядке.

```
> (setf (gethash 'shape ht) 'spherical
        (gethash 'size ht) 'giant)
GIANT
> (maphash #'(lambda (k v)
               (format t "~A = ~A~%" k v))
           ht)
SHAPE = SPHERICAL
SIZE = GIANT
COLOR = RED
NIL
```

Функция maphash всегда возвращает nil, однако вы можете сохранить данные, если передадите функцию, которая, например, будет накапливать результаты в списке.

Хеш-таблицы могут накапливать любое количество вхождений, так как способны расширяться в процессе работы, когда места для хранения элементов перестанет хватать. Задать исходную емкость таблицы можно с помощью ключа :size функции make-hash-table. Используя этот параметр, вам следует помнить о двух вещах: большой размер таблицы позволит избежать ее частых расширений (это довольно затратная процедура), однако создание таблицы большого размера для малого набора данных приведет к необоснованным затратам памяти. Важный момент:

параметр :size определяет не количество хранимых объектов, а количество пар ключ-значение. Выражение:

```
(make-hash-table :size 5)
```

вернет хеш-таблицу, которая сможет вместить пять вхождений до того, как ей придется расширяться.

Как и любая структура, подразумевающая возможность поиска элементов, хеш-таблица может использовать различные предикаты проверки эквивалентности. По умолчанию используется eql, но также можно применить eq, equal или equalp; используемый предикат указывается с помощью ключа :test:

```
> (setf writers (make-hash-table :test #'equal)
#<Hash-Table C005E6>
> (setf (gethash '(ralph waldo emerson) writers) t)
T
```

Это один из компромиссов, с которым нам приходится мириться ради получения эффективных хеш-таблиц. Работая со списками, мы бы воспользовались функцией member, которой можно каждый раз сообщать различные предикаты проверки. При работе с хеш-таблицами мы должны определиться с используемым предикатом заранее, в момент ее создания.

С необходимостью жертвовать одним ради другого в Лисп-разработке (да и в жизни в целом) вам придется столкнуться не раз. Это часть философии Лиспа: первоначально вы миритесь с низкой производительностью ради удобства разработки, а по мере развития программы можете пожертвовать ее гибкостью ради скорости выполнения.

Итоги главы

1. Common Lisp поддерживает массивы не менее семи размерностей. Одномерные массивы называются векторами.
2. Строки – это векторы знаков. Знаки – это полноценные самостоятельные объекты.
3. Последовательности включают в себя списки и векторы. Большинство функций для работы с последовательностями имеют аргументы по ключу.
4. Синтаксический разбор не составляет труда в Common Lisp благодаря наличию множества полезных функций для работы со строками.
5. Вызов defstruct определяет структуру с именованными полями. Это хороший пример программы, которая пишет другие программы.
6. Двоичные деревья поиска удобны для хранения отсортированного набора объектов.
7. Хеш-таблицы – это эффективный способ представления множеств и отображений.

Упражнения

1. Определите функцию, поворачивающую квадратный массив (массив с размерностями (n n)) на 90 градусов по часовой стрелке:

```
> (quarter-turn #2A((a b) (c d)))
#2A((C A) (D B))
```

   Вам потребуется `array-dimensions` (стр. 374).

2. Разберитесь с описанием `reduce` на стр. 382, затем с ее помощью определите:

   (a) `copy-list`
   (b) `reverse` (для списков)

3. Создайте структуру для дерева, каждый узел которого помимо некоторых данных имеет трех потомков. Определите:

   (a) функцию, копирующую такое дерево (каждый узел скопированного дерева не должен быть эквивалентен исходному с точки зрения `eql`);

   (b) функцию, принимающую объект и такое дерево и возвращающую истину, если этот объект встречается (с точки зрения `eql`) в поле данных хотя бы одного узла дерева.

4. Определите функцию, которая строит из BST-дерева список его объектов, отсортированный от большего к меньшему.

5. Определите `bst-adjoin`[1]. Функция работает аналогично `bst-insert`, однако добавляет новый объект лишь в том случае, если он отсутствует в имеющемся дереве.

6. Содержимое любой хеш-таблицы может быть представлено в виде ассоциативного списка с элементами (k . v) для каждой пары ключ-значение. Определите функцию, строящую:

   (a) хеш-таблицу по ассоциативному списку;

   (b) ассоциативный список по хеш-таблице.

---

[1] В действительности, определение `bst-adjoin` будет эквивалентно уже имеющемуся `bst-insert`. На это указал Ричард Грин. – *Прим. перев.*

5

Управление

В разделе 2.2 впервые упоминается порядок вычисления, который теперь должен быть вам хорошо знаком. В данной главе будут рассмотрены операторы, которые не подчиняются этому правилу, позволяя вам самостоятельно управлять ходом вычисления. Если обычные функции считать листьями Лисп-программы, то с помощью таких операторов можно создавать ее ветви.

5.1. Блоки

В Common Lisp есть три основных оператора для создания блоков кода: `progn`, `block` и `tagbody`. С первым из них мы уже знакомы. Выражения, составляющие тело оператора `progn`, вычисляются последовательно, при этом возвращается значение последнего:°

```
> (progn
    (format t "a")
    (format t "b")
    (+ 11 12))
ab
23
```

Так как возвращается значение лишь последнего выражения, то использование `progn` (или других операторов блоков) предполагает наличие побочных эффектов.

Оператор `block` – это `progn` с именем и запасным выходом. Первый аргумент должен быть символом, и он определяет имя блока. Находясь внутри блока, вы можете в любой момент возвратить значение с помощью `return-from` с соответствующим именем блока:

```
> (block head
    (format t "Here we go.")
```

```
    (return-from head 'idea)
    (format t "We'll never see this. "))
Here we go.
IDEA
```

Вызов return-from позволяет вам внезапно, но изящно выйти из любого места в коде. Второй аргумент return-from служит в качестве возвращаемого значения из блока, имя которого передается в первом аргументе. Все выражения, находящиеся в соответствующем блоке после return-from, не вычисляются.

Кроме того, существует специальный макрос return, выполняющий выход из блока с именем nil:

```
> (block nil
    (return 27))
27
```

Многие операторы в Common Lisp, принимающие на вход блок кода, неявно оборачивают его в block с именем nil. В частности, так делают все итерационные конструкции:

```
> (dolist (x '(a b c d e))
    (format t "~A " x)
    (if (eql x 'c)
        (return 'done)))
A B C
DONE
```

Тело функции, создаваемой defun, является блоком с тем же именем, что и сама функция:

```
(defun foo ()
  (return-from foo 27))
```

Вне явного или неявного block ни return, ни return-from не работают.

С помощью return-from можно написать улучшенный вариант функции read-integer:

```
(defun read-integer (str)
  (let ((accum 0))
    (dotimes (pos (length str))
      (let ((i (digit-char-p (char str pos))))
        (if i
            (setf accum (+ (* accum 10) i))
            (return-from read-integer nil))))
    accum))
```

Предыдущий вариант read-integer (стр. 83) выполнял проверку всех знаков до построения целого числа. Теперь же два шага собраны воедино, так как использование return-from позволяет прервать выполнение, когда мы встретим нечисловой знак.

Третий основной конструктор блоков, tagbody, допускает внутри себя использование оператора go. Атомы, встречающиеся внутри блока, рас-

цениваются как метки, по которым может выполняться переход. Ниже приведен довольно корявый код для печати чисел от 1 до 10:

```
> (tagbody
    (setf x 0)
   top
    (setf x (+ x 1))
    (format t "~A " x)
    (if (< x 10) (go top)))
1 2 3 4 5 6 7 8 9 10
NIL
```

Оператор tagbody – один из тех, которые применяются для построения других операторов, но, как правило, не годятся для непосредственного использования. Большинство итеративных операторов построены поверх tagbody, поэтому иногда (но не всегда) есть возможноть использовать внутри них метки и переход по ним с помощью go.

Как решить, какой из блоков использовать? Практически всегда подойдет progn. При желании иметь возможность экстренного выхода лучше использовать block. Практически никто не использует tagbody напрямую.

5.2. Контекст

С оператором let, позволяющим группировать выражения, мы уже знакомы. Помимо набора вычисляемых выражений он позволяет задавать новые переменные, которые действуют внутри его тела:

```
> (let ((x 7)
        (y 2))
    (format t "Number")
    (+ x y))
Number
9
```

Операторы типа let создают новый *лексический контекст* (*lexical context*). В нашем примере лексический контекст имеет две новые переменные, которые вне его становятся невидимыми.

Вызов let можно понимать как вызов функции. В разделе 2.14 показано, что на функцию можно ссылаться не только по имени, но и по лямбда-выражению. Раз лямбда-выражение эквивалентно имени функции, то мы можем использовать его вместо имени, ставя первым элементом выражения:

```
> ((lambda (x) (+ x 1)) 3)
4
```

Пример с let, приведенный в начале раздела, может быть переписан как лямбда-вызов:

```
((lambda (x y)
   (format t "Number")
```

```
    (+ x y))
  7
  2)
```

Любой вопрос, возникающий у вас при использовании let, легко разрешится, если построить аналогичную конструкцию с помощью lambda.°

Приведем пример ситуации, которую разъясняет наша аналогия. Рассмотрим выражение let, в котором одна из переменных зависит от другой переменной той же let-конструкции:

```
(let ((x 2)
      (y (+ x 1)))
  (+ x y))
```

Значение x в (+ x 1) не определено, и это видно из соответствующего лямбда-вызова:

```
((lambda (x y) (+ x y)) 2
                        (+ x 1))
```

Совершенно очевидно, что выражение (+ x 1) не может ссылаться на переменную x в лямбда-выражении.

Но как быть, если мы хотим, чтобы одна из переменных в let-вызове зависела от другой? Для этой цели существует оператор let*:

```
> (let* ((x 1)
         (y (+ x 1)))
    (+ x y))
3
```

Вызов let* полностью эквивалентен серии вложенных let. Наш пример можно переписать так:

```
(let ((x 1))
  (let ((y (+ x 1)))
    (+ x y)))
```

Как в let, так и в let* исходные значения переменных – nil. Если именно эти значения вам нужны, можно не заключать объявления в скобки:

```
> (let (x y)
    (list x y))
(NIL NIL)
```

Макрос destructing-bind является обобщением let. Вместо отдельных переменных он принимает *шаблон* (*pattern*) – одну или несколько переменных, расположенных в виде дерева, – и связывает эти переменные с соответствующими частями реального дерева. Например:

```
> (destructing-bind (w (x y) . z) '(a (b c) d e)
    (list w x y z))
(A B C (D E))
```

В случае несоответствия шаблона дереву, переданному во втором аргументе, возникает ошибка.

5.3. Условные выражения

Наиболее простым условным оператором можно считать if. Все остальные являются его расширениями. Оператор when является упрощением if. Его тело, которое может состоять из нескольких выражений, вычисляется лишь в случае истинности тестового выражения. Таким образом,

```
(when (oddp that)
  (format t "Hmm, that's odd. ")
  (+ that 1))
```

эквивалентно

```
(if (oddp that)
    (progn
      (format t "Hmm, that's odd. ")
      (+ that 1)))
```

Оператор unless действует противоположно when. Его тело вычисляется лишь в случае ложности тестового выражения.

Матерью всех условных выражений (в обоих смыслах) является cond, который предлагает два новых преимущества: неограниченное количество условных переходов и неявное использование progn в каждом из них. Его имеет смысл использовать в случае, когда третий аргумент if – другое условное выражение. Например, следующее определение our-member:

```
(defun our-member (obj lst)
  (if (atom lst)
      nil
      (if (eql (car lst) obj)
          lst
          (our-member obj (cdr lst)))))
```

может быть переписано с помощью cond:

```
(defun our-member (obj lst)
  (cond ((atom lst) nil)
        ((eql (car lst) obj) lst)
        (t (our-member obj (cdr lst)))))
```

В действительности, реализация Common Lisp, вероятно, оттранслирует второй вариант в первый.

В общем случае cond может принимать ноль или более аргументов. Каждый аргумент должен быть представлен списком, состоящим из условия и следующих за ним выражений, которые будут вычисляться в случае истинности этого условия. При вычислении вызова cond условия проверяются по очереди до тех пор, пока одно из них не окажется истинным. Далее вычисляются выражения, следующие за этим условием, и возвращается значение последнего из них. Если после выражения, оказавшегося истинным, нет других выражений, то возвращается его значение:

```
> (cond (99))
99
```

Если условие имеет тестовое выражение t, оно будет истинным всегда. Этот факт удобно использовать для указания условия, которое будет выполняться в случае, когда все остальные не подошли. По умолчанию в таком случае возвращается nil, но считается плохим стилем использовать этот вариант. (Подобная проблема показана на стр. 132.)

Если вы сравниваете выражение с набором постоянных, уже вычисленных объектов, правильнее будет использовать конструкцию case. Например, мы можем воспользоваться case для преобразования имени месяца в его продолжительность в днях:

```
(defun month-length (mon)
  (case mon
    ((jan mar may jul aug oct dec) 31)
    ((apr jun sept nov) 30)
    (feb (if (leap-year) 29 28))
    (otherwise "unknown month")))
```

Конструкция case начинается с выражения, значение которого будет сравниваться с предложенными вариантами. За ним следуют выражения, начинающиеся либо с ключа, либо со списка ключей, за которым следует произвольное количество выражений. Сами ключи рассматриваются как константы и не вычисляются. Значение ключа (ключей) сравнивается с оригинальным объектом с помощью eql. В случае совпадения вычисляются следующие за ключом выражения. Результатом вызова case является значение последнего вычисленного выражения.

Вариант по умолчанию может быть обозначен через t или otherwise. Если ни один из ключей не подошел или вариант, содержащий подходящий ключ, не содержит выражений, возвращается nil:

```
> (case 99 (99))
NIL
```

Макрос typecase похож на case, но вместо ключей использует спецификаторы типов, а искомое выражение сверяется с ними через typep вместо eql. (Пример с использованием typecase приведен на стр. 118.)

5.4. Итерации

Основной итерационный оператор, do, был представлен в разделе 2.13. Поскольку do неявным образом использует block и tagbody, внутри него можно использовать return, return-from и go.

В разделе 2.13 было указано, что первый аргумент do должен быть списком, содержащим описания переменных вида

(*variable initial update*)

Выражения *initial* и *update* не обязательны. Если выражение *update* пропущено, значение соответствующей переменной не будет меняться. Если пропущено *initial*, то переменной присваивается первоначальное значение `nil`.

В примере на стр. 40:

```
(defun show-squares (start end)
  (do ((i start (+ i 1)))
      ((> i end) 'done)
    (format t "~A ~A~%" i (* i i))))
```

для переменной, создаваемой `do`, задано выражение *update*. Практически всегда в процессе выполнения выражения `do` изменяется хотя бы одна переменная.

В случае модификации сразу нескольких переменных встает вопрос: что произойдет, если форма обновления одной из них ссылается на другую переменную, которая также должна обновиться? Какое значение последней переменной будет использоваться: зафиксированное на предыдущей итерации или уже измененное? Макрос `do` использует значение предыдущей итерации:

```
> (let ((x 'a))
    (do ((x 1 (+ x 1))
         (y x x))
        ((> x 5))
      (format t "(~A ~A) " x y)))
(1 A) (2 1) (3 2) (4 3) (5 4)
NIL
```

На каждой итерации значение `x` увеличивается на `1`, а `y` получает значение `x` из *предыдущей* итерации.

Существует также оператор `do*`, относящийся к `do`, как `let*` к `let`. В нем начальные (*initial*) или последующие (*update*) выражения для вычисления переменных могут ссылаться на предыдущие переменные, и при этом они будут получать их текущие значения:

```
> (do* ((x 1 (+ x 1))
        (y x x))
       ((> x 5))
    (format t "(~A ~A) " x y))
(1 1) (2 2) (3 3) (4 4) (5 5)
NIL
```

Помимо `do` и `do*` существуют и более специализированные операторы. Итерация по списку выполняется с помощью `dolist`:

```
> (dolist (x '(a b c d) 'done)
    (format t "~A " x))
A B C D
DONE
```

Суть оператора do

В книге «The Evolution of Lisp» Стил и Гэбриэл выражают суть do настолько точно, что соответствующий отрывок из нее достоин цитирования:

Отложим обсуждение синтаксиса в сторону. Нужно признать, что цикл, выполняющий лишь итерацию по одной переменной, довольно бесполезен в любом языке программирования. Практически всегда в таких случаях одна переменная используется для получения последовательных значений, в то время как еще одна собирает их вместе. Если цикл лишь увеличивает значение переменной на заданную величину, то каждое новое значение должно устанавливаться благодаря присваиванию... или иным побочным эффектам. Цикл do с несколькими переменными выражает внутреннюю симметрию генерации и аккумуляции, позволяя выразить итерацию без явных побочных эффектов:

```
(defun factorial (n)
  (do ((j n (- j 1))
       (f 1 (* j f)))
      ((= j 0) f)))
```

И вправду, нет ничего необычного в циклах do с пустым телом, которые выполняют всю работу в самих *шагах* итерации.°

Третье выражение в начальном списке будет вычислено на выходе из цикла. По умолчанию это nil.

Похожим образом действует dotimes, пробегающий для заданного числа n значения от 0 до n–1:

```
> (dotimes (x 5 x)
    (format t "~A " x))
0 1 2 3 4
5
```

Как и в dolist, третье выражение в начальном списке не обязательно и по умолчанию установлено как nil. Обратите внимание, что это выражение может содержать саму итерируемую переменную.

Функция mapc похожа на mapcar, однако она не строит список из вычисленных значений, поэтому может использоваться лишь ради побочных эффектов. Она более гибка, чем dolist, потому что может выполнять итерацию параллельно сразу по нескольким спискам:

```
> (mapc #'(lambda (x y)
            (format t "~A ~A " x y))
        '(hip flip slip)
        '(hop flop slop))
```

```
HIP HOP FLIP FLOP SLIP SLOP
(HIP FLIP SLIP)
```

Функция mapc всегда возвращает свой второй аргумент.

5.5. Множественные значения

Чтобы подчеркнуть важность функционального программирования, часто говорят, что в Лиспе каждое выражение возвращает значение. На самом деле, все несколько сложнее. В Common Lisp выражение может возвращать несколько значений или же не возвращать ни одного. Максимальное их количество может различаться в разных реализациях, но должно быть не менее 19.

Для функции, которая вычисляет несколько значений, эта возможность позволяет возвращать их без заключения в какую-либо структуру. Например, встроенная функция get-decoded-time возвращает текущее время в виде девяти значений: секунды, минуты, часы, день, месяц, год и еще два.

Множественные значения также позволяют функциям поиска разделять случаи, когда элемент не найден и когда найден элемент nil. Именно поэтому gethash возвращает два значения. *Второе* значение информирует об успешности поиска, поэтому мы можем хранить nil в хеш-таблице точно так же, как и остальные значения.

Множественные значения можно возвратить с помощью функции values. Она возвращает в точности те значения, которые были переданы ей в виде аргументов:

```
> (values 'a nil (+ 2 4))
A
NIL
6
```

Если вызов values является последним в теле функции, то эта функция возвращает те же множественные значения, что и values. Множественные значения могут передаваться через несколько вызовов:

```
> ((lambda () ((lambda () (values 1 2)))))
1
2
```

Однако если в цепочке передачи значений что-либо принимает лишь один аргумент, то все остальные будут потеряны:

```
> (let ((x (values 1 2)))
    x)
1
```

Используя values без аргументов, мы имеем возможность не возвращать никаких значений. Если же что-либо будет ожидать значение, будет возвращен nil.

```
> (values)
> (let ((x (values)))
    x)
NIL
```

Чтобы получить несколько значений, можно воспользоваться `multiple-value-bind`:

```
> (multiple-value-bind (x y z) (values 1 2 3)
     (list x y z))
(1 2 3)
> (multiple-value-bind (x y z) (values 1 2)
    (list x y z))
(1 2 NIL)
```

Если переменных больше, чем возвращаемых значений, лишним переменным будет присвоено значение `nil`. Если же значений возвращается больше, чем выделено переменных, то лишние значения будут отброшены. Таким образом, вы можете написать функцию, которая просто печатает текущее время:°

```
> (multiple-value-bind (s m h) (get-decoded-time)
    (format nil "~A:~A:~A" h m s))
"4:32:13"
```

Можно передать какой-либо функции множественные значения в качестве аргументов с помощью `multiple-value-call`:

```
> (multiple-value-call #'+ (values 1 2 3))
6
```

Кроме того, есть функция `multiple-value-list`:

```
> (multiple-value-list (values 'a 'b 'c))
(A B C)
```

которая действует так же, как `multiple-value-call` с функцией `#'list` в качестве первого аргумента.

5.6. Прерывание выполнения

Чтобы выйти из блока в любой момент, достаточно вызвать `return`. Однако иногда может потребоваться нечто более радикальное, например передача управления через несколько вызовов функций. Для этой цели существуют специальные операторы `catch` и `throw`. Конструкция `catch` использует метку, которой может быть любой объект. За меткой следует набор выражений.

```
(defun super ()
  (catch 'abort
    (sub)
    (format t "We'll never see this.")))
```

```
(defun sub ()
  (throw 'abort 99))
```

Выражения после метки вычисляются так же, как в progn. Вызов throw внутри любого из этих выражений приведет к немедленному выходу из catch:

```
> (super)
99
```

Оператор throw с заданной меткой передаст управление соответствующей конструкции catch (то есть завершит ее выполнение) через любое количество catch с другими метками (соответственно «убив» их). Если нет ожидающего catch с требуемой меткой, throw вызовет ошибку.

Вызов error также прерывает выполнение, но вместо передачи управления вверх по дереву вызовов он передает его обработчику ошибок Лиспа. В результате вы, скорее всего, попадете в отладчик (break loop). Вот что произойдет в некой абстрактной реализации Common Lisp:

```
> (progn
    (error "Oops! ")
    (format t "After the error. "))
Error: Oops!
       Options: :abort, :backtrace
>>
```

Более подробную информацию об ошибках и условиях можно найти в приложении А.

Иногда вам может понадобиться некая гарантия защиты от ошибок и прерываний типа throw. Используя unwind-protect, вы можете быть уверены, что в результате подобных явлений программа не окажется в противоречивом состоянии. Конструкция unwind-protect принимает любое количество аргументов и возвращает значение первого. Остальные выражения будут вычислены, даже если вычисление первого будет прервано.

```
> (setf x 1)
1
> (catch 'abort
    (unwind-protect
      (throw 'abort 99)
      (setf x 2)))
99
> x
2
```

Здесь, даже несмотря на то что throw передал управление catch, второе выражение было вычислено прежде, чем был осуществлен выход из catch. В случае когда перед преждевременным завершением необходима какая-то очистка или сброс, unwind-protect может быть очень полезна. Один из таких примеров представлен на стр. 295.

5.7. Пример: арифметика над датами

В некоторых приложениях полезно иметь возможность складывать и вычитать даты, например, чтобы сказать, что через 60 дней после 17 декабря 1997 года настанет 15 февраля 1998. В этом разделе мы создадим необходимые для этого инструменты. Нам потребуется конвертировать даты в целые числа, при этом за ноль будет принята дата 1 января 2000 года. Затем мы сможем работать с датами как с целыми числами, используя функции + и –. Кроме того, необходимо уметь конвертировать целое число обратно в дату.

Чтобы преобразовать дату в целое число, будем складывать количества дней, соответствующие различным компонентам даты. Например, число, соответствующее 13 ноября 2004 года, получим, складывая количество дней до 2004 года, количество дней в году до ноября и число 13.

Нам потребуется таблица, устанавливающая соответствие между месяцем и количеством дней в нем для невисокосного года. Начнем со списка, содержащего длины месяцев:

```
> (setf mon '(31 28 31 30 31 30 31 31 30 31 30 31))
(31 28 31 30 31 30 31 31 30 31 30 31)
```

Проведем несложную проверку, сложив все эти числа:

```
> (apply #'+ mon)
365
```

Теперь перевернем этот список и применим с помощью `maplist` функцию + к последовательности хвостов (`cdr`) списка. Мы получим количества дней, прошедшие до начала каждого последующего месяца:

```
> (setf nom (reverse mon))
(31 30 31 30 31 31 30 31 30 31 28 31)
> (setf sums (maplist #'(lambda (x)
                          (apply #'+ x))
                      nom))
(365 334 304 273 243 212 181 151 90 59 31)
> (reverse sums)
(31 59 90 120 151 181 212 243 273 304 334 365)
```

Эти цифры означают, что до начала февраля пройдет 31 день, до начала марта 59 и т. д.

На рис. 5.1. приведен код, выполняющий преобразования дат. В нем полученный нами список преобразован в вектор.

Жизненный цикл обычной Лисп-программы состоит из четырех этапов: сначала она пишется, потом читается, компилируется и затем исполняется. Отличительной особенностью Лиспа является тот факт, что Лисп-система принимает участие в каждом шаге этой последовательности. Вы можете взаимодействовать с Лиспом не только во время работы программы, но также во время компиляции (раздел 10.2) и даже ее чтения

(раздел 14.3). То, каким образом мы создали вектор month, свидетельствует о том, что мы используем Лисп даже во время написания программы.

Производительность программы имеет значение лишь на четвертом этапе – при ее выполнении. На первых трех этапах вы можете в полной мере пользоваться гибкостью и мощью списков, не задумываясь об их стоимости.

```
(defconstant month
  #(0 31 59 90 120 151 181 212 243 273 304 334 365))

(defconstant yzero 2000)

(defun leap? (y)
  (and (zerop (mod y 4))
       (or (zerop (mod y 400))
           (not (zerop (mod y 100))))))

(defun date->num (d m y)
  (+ (- d 1) (month-num m y) (year-num y)))

(defun month-num (m y)
  (+ (svref month ( - m 1))
     (if (and (> m 2) (leap? y)) 1 0)))

(defun year-num (y)
  (let ((d 0))
    (if (>= y yzero)
        (dotimes (i (- y yzero) d)
          (incf d (year-days (+ yzero i))))
        (dotimes (i (- yzero y) (- d))
          (incf d (year-days (+ y i)))))))

(defun year-days (y) (if (leap? y) 366 365))
```

Рис. 5.1. *Операции с датами: преобразование дат в целые числа*

Если вы вздумаете использовать этот код для управления машиной времени, то люди могут не согласиться с вами по поводу текущей даты, если вы попадете в прошлое. По мере накопления информации о продолжительности года люди вносили изменения в календарь. В англоговорящих странах подобная нестыковка последний раз устранялась в 1752 году, когда сразу после 2 сентября последовало 14 сентября.°

Количество дней в году зависит от того, является ли он високосным. Год не является високосным, если он не кратен 4 либо кратен 100 и не кратен 400. 1904 год был високосным, 1900 не был, а 1600 был.

Для определения делимости существует функция mod, возвращающая остаток от деления первого аргумента на второй:

```
> (mod 23 5)
3
```

```
> (mod 25 5)
0
```

Одно число считается делимым на другое, если остаток от деления на него равен нулю. Функция leap? использует этот подход для определения високосности года:

```
> (mapcar #'leap? '(1904 1900 1600))
(T NIL T)
```

Дату в целое число преобразует функция date->num. Она возвращает сумму численных представлений каждого компонента даты. Чтобы выяснить, сколько дней прошло до начала заданного месяца, она использует функцию month-num, которая обращается к соответствующему компоненту вектора month, добавляя к нему 1, если год високосный и заданный месяц находится после февраля.

Чтобы найти количество дней до наступления заданного года, date->num вызывает year-num, которая возвращает численное представление 1 января этого года. Эта функция отсчитывает годы до нулевого года (то есть 2000).

На рис. 5.2 показан код второй части программы. Функция num->date преобразует целые числа обратно в даты. Вызывая num-year, она получает год и количество дней в остатке, по которому num-month вычисляет месяц и день.

Как и year-num, num-year ведет отсчет от 2000 года, складывая продолжительности каждого года до тех пор, пока эта сумма не превысит заданное число n (или не будет равна ему). Если сумма, полученная на какой-то итерации, превысит его, то num-year возвращает год, полученный на предыдущей итерации. Для этого введена переменная prev, которая хранит число дней, полученное на предыдущей итерации.

Функция num-month и ее подпроцедура nmon ведут себя обратно month-num. Они вычисляют позицию в векторе month по численному значению, тогда как month-num вычисляет значение исходя из заданного элемента вектора.

Первые две функции на рис. 5.2 могут быть объединены в одну. Вместо вызова отдельной функции num-year могла бы вызывать непосредственно num-month. Однако на этапе тестирования и отладки текущий вариант более удобен, и объединение функций может стать следующим шагом после тестирования.

Функции date->num и num->date упрощают арифметику дат.° С их помощью работает функция date+, позволяющая складывать и вычитать дни из заданной даты. Теперь мы сможем вычислить дату по прошествии 60 дней с 17 декабря 1997 года:

```
> (multiple-value-list (date+ 17 12 1997 60))
(15 2 1998)
```

Мы получили 15 февраля 1998 года.

```
(defun num->date (n)
  (multiple-value-bind (y left) (num-year n)
    (multiple-value-bind (m d) (num-month left y)
      (values d m y))))

(defun num-year (n)
  (if (< n 0)
      (do* ((y (- yzero 1) (- y 1))
            (d (- (year-days y)) (- d (year-days y))))
           ((<= d n) (values y (- n d))))
      (do* ((y yzero (+ y 1))
            (prev 0 d)
            (d (year-days y) (+ d (year-days y))))
           ((> d n) (values y (- n prev))))))

(defun num-month (n y)
  (if (leap? y)
      (cond ((= n 59) (values 2 29))
            ((> n 59) (nmon (- n 1)))
            (t        (nmon n)))
      (nmon n)))

(defun nmon (n)
  (let ((m (position n month :test #'<)))
    (values m (+ 1 (- n (svref month (- m 1)))))))

(defun date+ (d m y n)
  (num->date (+ (date->num d m y) n)))
```

Рис. 5.2. *Операции с датами: преобразование целых чисел в даты*

Итоги главы

1. В Common Lisp есть три основных конструкции для блоков: `progn`, `block` с возможностью немедленного выхода (`return`) и `tagbody`, внутри которой работает goto. Многие встроенные операторы используют неявные блоки.
2. Создание нового лексического контекста эквивалентно вызову функции.
3. В Common Lisp есть набор условных операторов, приспособленных для различных ситуаций. Все они могут быть определены с помощью `if`.
4. В Common Lisp есть также разнообразные итерационные операторы.
5. Выражения могут возвращать несколько значений.
6. Вычисления могут быть прерваны, а также могут быть защищены от последствий таких прерываний.

Упражнения

1. Запишите следующие выражения без использования `let` или `let*`, а также без вычисления одного и того же выражения дважды:

   (a)
   ```
   (let ((x (car y)))
     (cons x x))
   ```
   (b)
   ```
   (let* ((w (car x))
          (y (+ w z)))
     (cons w y))
   ```

2. Перепишите функцию `mystery` (стр. 46) с использованием `cond`.
3. Определите функцию, возвращающую квадрат своего аргумента, лишь когда аргумент – положительное число, меньшее или равное пяти.
4. Перепишите `month-num` (рис. 5.1), используя `case` вместо `svref`.
5. Определите функцию (рекурсивную и итеративную версии) от объекта x и вектора v, возвращающую список объектов, следующих непосредственно перед x в v:

   ```
   > (precedes #\a "abracadabra")
   (#\c #\d #\r)
   ```

6. Определите функцию (итеративно и рекурсивно), принимающую объект и список и возвращающую новый список, в котором заданный элемент находится между каждой парой элементов исходного списка:

   ```
   > (intersperse '- '(a b c d))
   (A - B - C - D)
   ```

7. Определите функцию, принимающую список чисел и возвращающую истину, если разница между каждой последующей их парой равна `1`. Используйте:

   (a) рекурсию

   (b) `do`

   (c) `mapc` и `return`

8. Определите одиночную рекурсивную функцию, которая возвращает два значения – максимальный и минимальный элементы вектора.
9. Программа на рис. 3.12 продолжает поиск после нахождения первого подходящего пути в очереди. Для больших сетей это может стать проблемой.

   (a) Используя `catch` и `throw`, измените программу таким образом, чтобы она возвращала первый найденный путь сразу же после того, как он найден.

   (b) Перепишите программу, чтобы она делала то же самое без использования `catch` и `throw`.

6

Функции

Понимание функций – один из ключей к пониманию всего Лиспа. Функции – основная концепция, которая легла в основу этого языка. На практике они являются наиболее полезным инструментом, находящимся в вашем распоряжении.

6.1. Глобальные функции

Предикат fboundp сообщает, существует ли функция с именем, заданным в виде символа. Если какой-либо символ является именем функции, то ее можно получить с помощью symbol-function:

```
> (fboundp '+)
T
> (symbol-function '+)
#<Compiled-Function + 17BA4E>
```

Используя setf над symbol-function:

```
> (setf (symbol-function 'add2)
        #'(lambda (x) (+ x 2)))
```

мы можем определить новую глобальную функцию, которую можно использовать точно так же, как если бы мы определили ее с помощью defun:

```
> (add2 1)
3
```

Фактически defun делает немногим более чем просто преобразование выражения типа

```
(defun add2 (x) (+ x 2))
```

в выражение с setf. Использование defun делает программы более легкими для понимания, а также сообщает компилятору некоторую ин-

формацию о функции, хотя, строго говоря, использовать defun при написании программ вовсе не обязательно.

Сделав первым аргументом defun выражение вида (setf *f*), вы определите, что будет происходить, когда setf вызван с первым аргументом *f*. Ниже приведен пример функции primo, которая является аналогом car:

```
(defun primo (lst) (car lst))

(defun (setf primo) (val lst)
  (setf (car lst) val))
```

В последнем определении на месте имени функции находится выражение (setf primo), первым параметром является устанавливаемое значение, а вторым – аргумент primo.

Теперь любой setf для primo будет являться вызовом функции, определенной выше:

```
> (let ((x (list 'a 'b 'c)))
    (setf (primo x) 480)
    x)
(480 B C)
```

Совсем не обязательно определять саму функцию primo, чтобы определить поведение (setf primo), однако такие определения обычно даются парами.

Поскольку строки в Лиспе – полноценные выражения, ничто не мешает им находиться внутри тела функции. Сама по себе строка не вызывает побочных эффектов, поэтому она ни на что не влияет, если не является последним выражением. Если же строка будет первым выражением, она будет считаться строкой документации к функции:

```
(defun foo (x)
  "Implements an enhanced paradigm of diversity."
  x)
```

Документация к функции, определенной глобально, может быть получена с помощью documentation:

```
> (documentation 'foo 'function)
"Implements an enhanced paradigm of diversity."
```

6.2. Локальные функции

Функции, определенные с помощью defun или setf вместе с symbol-function, являются *глобальными*. Как и глобальные переменные, они могут быть использованы везде. Кроме того, есть возможность определять *локальные* функции, которые, как и локальные переменные, доступны лишь внутри определенного контекста.

Локальные функции могут быть определены с помощью конструкции labels – своего рода let для функций. Ее первым аргументом является не

список, задающий переменные, а список определений локальных функций. Каждый элемент этого списка является списком следующего вида:

(*name parameters . body*)

Внутри `labels` любое выражение с *name* эквивалентно лямбда-вызову (`lambda` *parameters . body*).

```
> (labels ((add10 (x) + x 10))
          (consa (x) (cons 'a x)))
    (consa (add10 3)))
(A . 13)
```

Аналогия с `let`, однако, не работает в одном случае. Локальная функция в `labels` может ссылаться на любые другие функции, определенные в той же конструкции `labels`, в том числе и на саму себя. Это позволяет создавать с помощью `labels` локальные рекурсивные функции:

```
> (labels ((len (lst)
           (if (null lst)
               0
               (+ (len (cdr lst)) 1))))
    (len '(a b c)))
3
```

В разделе 5.2 говорилось о том, что конструкция `let` может рассматриваться как вызов функции. Аналогично выражение с `do` может считаться вызовом рекурсивной функции. Выражение:

```
(do ((x a (b x))
     (y c (d y)))
    ((test x y) (z x y))
  (f x y))
```

эквивалентно

```
(labels ((rec (x y)
          (cond ((test x y)
                 (z x y))
                (t
                 (f x y)
                 (rec (b x) (d y))))))
  (rec a c))
```

Эта модель может служить для разрешения любых вопросов, которые у вас еще могли остаться касательно поведения `do`.

6.3. Списки параметров

В разделе 2.1 было показано, что благодаря префиксной нотации + может принимать любое количество аргументов. С тех пор мы познакомились с многими подобными функциями. Чтобы написать такую функцию самостоятельно, нам придется воспользоваться *остаточным* параметром (*rest*).

Если мы поместим элемент &rest перед последней переменной в списке параметров функции, то при вызове этой функции последний параметр получит список всех оставшихся аргументов. Теперь мы можем написать funcall с помощью apply:

```
(defun our-funcall (fn &rest args)
  (apply fn args))
```

Также нам уже знакомы параметры, которые могут быть пропущены и в таком случае получат значение по умолчанию. Такие параметры называются *необязательными* (*optional*). (Обычные параметры иногда называются *обязательными* (*required*).) Если символ &optional встречается в списке параметров функции

```
(defun philosoph (thing &optional property)
  (list thing 'is property))
```

то все последующие аргументы будут необязательными, а их значением по умолчанию будет nil:

```
> (philosoph 'death)
(DEATH IS NIL)
```

Мы можем задать исходное значение явно, заключив его в скобки вместе с параметром. Следующая версия philosoph

```
(defun philosoph (thing &optional (property 'fun))
  (list thing 'is property))
```

имеет более «радостное» значение по умолчанию:

```
> (philosoph 'death)
(DEATH IS FUN)
```

Значением по умолчанию необязательного параметра может быть не только константа, но и любое Лисп-выражение. Оно будет вычисляться заново каждый раз, когда это потребуется.

Параметры по ключу (*keyword*) предоставляют большую гибкость, чем необязательные параметры. Поместив символ &key в список параметров, вы помечаете все последующие параметры как необязательные. Кроме того, при вызове функции эти параметры будут передаваться не в соответствии с их ожидаемым положением, а по соответствующей метке – ключевому слову:

```
> (defun keylist (a &key x y z)
    (list a x y z))
KEYLIST
> (keylist 1 :y 2)
(1 NIL 2 NIL)
> (keylist 1 :y 3 :x 2)
(1 2 3 NIL)
```

Для параметров по ключу, так же как и для необязательных, значением по умолчанию является nil, но в то же время можно задать выражение для их вычисления.

Ключевые слова и связанные с ними значения могут собираться с помощью &rest и в таком виде передаваться другим функциям, ожидающим их. Например, мы могли бы определить adjoin следующим образом:

```
(defun our-adjoin (obj lst &rest args)
  (if (apply #'member obj lst args)
      lst
      (cons obj lst)))
```

Так как adjoin принимает те же параметры по ключу, что и member, достаточно просто собрать их в список и передать member.

В разделе 5.2 был представлен макрос destructing-bind. В общем случае каждое поддерево шаблона, заданного первым аргументом, может быть устроено так же комплексно, как и список аргументов функции:

```
> (destructing-bind ((&key w x) &rest y) '((:w 3) a)
    (list w x y))
(3 NIL (A))
```

6.4. Пример: утилиты

В разделе 2.6 упоминалось, что Лисп состоит по большей части из таких же функций, как те, что вы можете определить самостоятельно. Эта особенность крайне полезна, потому что такой язык программирования не только не требует подгонки задачи под себя, но и сам может быть подстроен под каждую задачу. Если вы захотите видеть в Common Lisp какую-либо конкретную функцию, вы можете написать ее самостоятельно, и она станет такой же частью языка, как + или eql.

Опытные Лисп-разработчики создают программы как снизу-вверх, так и сверху-вниз. Подстраивая конкретную задачу под язык, они в то же время модифицируют язык, делая его удобнее для этой задачи. Таким образом, рано или поздно язык и задача окажутся максимально подогнанными друг под друга.

Функции, создаваемые для расширения возможностей Лиспа, называют *утилитами*. Создавая Лисп-программы, вы обнаружите, что некоторые функции, созданные для одной программы, могут пригодиться и для другой.

Профессиональные разработчики используют эту довольно привлекательную идею для создания повторно используемых программ. Этот подход некоторым образом связан с объектно-ориентированным программированием, однако программа вовсе не обязана быть объектно-ориентированной, чтобы быть пригодной к повторному использованию. Подтверждение тому – сами языки программирования (точнее их компиляторы), которые являются, вероятно, наиболее приспособленными к повторному использованию.

Основной секрет в написании таких программ – подход снизу-вверх, и программы вовсе не обязаны быть объектно-ориентированными,

чтобы писаться снизу-вверх. На самом деле, функциональный стиль даже лучше адаптирован для создания повторно используемых программ. Посмотрим, например, на sort. Имея в распоряжении эту функцию, вам вряд ли потребуется писать свои сортирующие процедуры на Лиспе, потому что sort эффективна и может быть приспособлена к самым разным задачам. Именно *это* и называется повторным использованием.

Вы можете применять этот подход в своих программах, создавая утилиты. На рис. 6.1 представлена небольшая выборка полезных утилит. Первые две, single? и append1, приведены с целью показать, что даже очень короткие утилиты могут быть полезными. Первая из них возвращает истину, когда ее аргумент – список из одного элемента:

```
> (single? '(a))
T
```

```
(defun single? (lst)
  (and (consp lst) (null (cdr lst))))

(defun append1 (lst obj)
  (append lst (list obj)))

(defun map-int (fn n)
  (let ((acc nil))
    (dotimes (i n)
      (push (funcall fn i) acc))
    (nreverse acc)))

(defun filter (fn lst)
  (let ((acc nil))
    (dolist (x lst)
      (let ((val (funcall fn x)))
        (if val (push val acc))))
    (nreverse acc)))

(defun most (fn lst)
  (if (null lst)
      (values nil nil)
      (let* ((wins (car lst))
             (max (funcall fn wins)))
        (dolist (obj (cdr lst))
          (let ((score (funcall fn obj)))
            (when (> score max)
              (setf wins obj
                    max  score))))
        (values wins max))))
```

Рис. 6.1. *Утилиты*

Вторая утилита напоминает cons, однако, в отличие от cons, она добавляет элемент в конец списка, а не в начало:

```
> (append1 '(a b c) 'd)
(A B C D)
```

Следующая утилита, map-int, принимает функцию и целое число n, возвращая список, состоящий из значений этой функции для всех целых чисел от 0 до n–1.

Такая функция может пригодиться при тестировании кода. (Одно из преимуществ Лиспа заключается в интерактивности разработки, благодаря которой вы можете создавать одни функции для тестирования других.) Если нам вдруг понадобится список чисел от 0 до 9, мы сможем написать:

```
> (map-int #'identity 10)
(0 1 2 3 4 5 6 7 8 9)
```

А если мы хотим получить список из десяти случайных чисел между 0 и 99 (включительно), просто пропустим параметр лямбда-выражения и напишем:

```
> (map-int #'(lambda (x) (random 100))
           10)
(85 40 73 64 28 21 67 5 32)
```

На примере map-int показана распространенная Лисп-идиома для построения списков. В ней создается аккумулятор acc, который исходно содержит nil, и в него последовательно добавляются элементы с помощью push. По завершении возвращается перевернутый список acc.[1]

Эту же идиому мы видим и в filter. Функция filter принимает функцию и список и возвращает список результатов применения функции к элементам исходного списка, причем возвращаемый список состоит только из элементов, отличающихся от nil:

```
> (filter #'(lambda (x)
              (and (evenp x) (+ x 10)))
          '(1 2 3 4 5 6 7))
(12 14 16)
```

Функцию filter можно также рассматривать как обобщение remove-if.

Последняя функция на рис. 6.1, most, возвращает элемент списка, имеющий наивысший рейтинг с точки зрения заданной рейтинговой функции. Помимо этого элемента она также возвращает соответствующее ему значение.

```
> (most #'length '((a b) (a b c) (a)))
(A B C)
3
```

[1] В данном контексте nreverse (стр. 229) делает то же, что и reverse, однако она более эффективна.

Если таких элементов несколько, `most` возвращает первый из них.

Обратите внимание, что последние три функции на рис. 6.1 принимают другие функции в качестве аргументов. Для Лиспа это явление совершенно естественно и является одной из причин его приспособленности к программированию снизу-вверх.° Хорошая утилита должна быть как можно более обобщенной, а абстрагировать общее легче, когда можно передать специфическую часть в качестве аргумента-функции.

Функции, приведенные в этом разделе, являются утилитами общего назначения и могут быть использованы в самых разных программах. Однако вы можете написать утилиты для более конкретных задач. Более того, далее будет показано, как создавать поверх Лиспа специализированные языки для конкретных прикладных областей. Это еще один хороший подход к получению кода для повторного использования.

6.5. Замыкания

Функция, как и любой другой объект, может возвращаться как результат выражения. Ниже приведен пример функции, которая возвращает функцию, применимую для сочетания объектов того же типа, что и ее аргумент:

```
(defun combiner (x)
  (typecase x
    (number #'+)
    (list   #'append)
    (t      #'list)))
```

С ее помощью мы сможем создать комбинирующую функцию общего вида:

```
(defun combine (&rest args)
  (apply (combiner (car args))
         args))
```

Она принимает аргументы любого типа и объединяет их в соответствии с типом. (Для упрощения рассматривается лишь тот случай, когда все аргументы имеют один тип.)

```
> (combine 2 3)
5
> (combine '(a b) '(c d))
(A B C D)
```

В разделе 2.10 объяснялось, что лексические переменные действительны только внутри контекста, в котором они были определены. Вместе с этим ограничением мы получаем обещание, что они будут *по-прежнему* действительны до тех пор, пока этот контекст где-нибудь используется.

Если функция была определена в зоне действия лексической переменной, то она сможет продолжать ссылаться на эту переменную, даже будучи возвращенной как значение, а затем использованной вне контекста

этой переменной. Приведем функцию, добавляющую 3 к своему аргументу:

```
> (setf fn (let ((i 3))
             #'(lambda (x) (+ x i))))
#<Interpreted-Function C0A51E>
> (funcall fn 2)
5
```

Если функция ссылается на определенную вне нее переменную, то такая переменная называется *свободной*. Функцию, ссылающуюся на свободную лексическую переменную, принято называть *замыканием* (*closure*)[1]. Свободная переменная будет существовать до тех пор, пока доступна использующая ее функция.

Замыкание – это комбинация функции и окружения. Замыкания создаются неявно, когда функция ссылается на что-либо из лексического окружения, в котором она была определена. Это происходит без каких-либо внешних проявлений, как, например, в функции, приведенной ниже:

```
(defun add-to-list (num lst)
  (mapcar #'(lambda (x)
              (+ x num))
          lst))
```

Функция принимает число и список и возвращает новый список, в котором к каждому элементу исходного списка добавляется заданное число. Переменная `num` в лямбда-выражении является свободной, значит, функции `mapcar` передается замыкание.

Более очевидным примером является функция, возвращающая само замыкание. Следующая функция возвращает замыкание, выполняющее сложение с заданным числом:

```
(defun make-adder (n)
  #'(lambda (x)
      (+ x n)))
```

Принимается число `n` и возвращается функция, которая складывает число `n` с *ее* аргументом:

```
> (setf add3 (make-adder 3))
#<Interpreted-Function C0EBF6>
> (funcall add3 2)
5
> (setf add27 (make-adder 27))
#<Interpreted-Function C0EE4E>
> (funcall add27 2)
29
```

[1] Название «замыкание» взято из ранних диалектов Лиспа. Оно происходит из метода, с помощью которого замыкания реализовывались в динамическом окружении.

Более того, несколько замыканий могут использовать одну и ту же переменную. Ниже определены две функции, использующие переменную counter:

```
(let ((counter 0))
  (defun reset ()
    (setf counter 0))
  (defun stamp ()
    (setf counter (+ counter 1))))
```

Такая пара функций может использоваться для создания меток времени. При каждом вызове stamp мы получаем число, на единицу большее, чем предыдущее. Вызывая reset, мы обнуляем счетчик:

```
> (list (stamp) (stamp) (reset) (stamp))
(1 2 0 1)
```

Несомненно, можно сделать то же самое с использованием глобальной переменной, однако последняя не защищена от воздействий извне.

В Common Lisp существует встроенная функция complement, принимающая предикат и возвращающая противоположный ему. Например:

```
> (mapcar (complement #'oddp)
          '(1 2 3 4 5 6))
(NIL T NIL T NIL T)
```

Благодаря замыканиям написать такую функцию очень просто:

```
(defun our-complement (f)
  #'(lambda (&rest args)
      (not (apply f args))))
```

Если вы отвлечетесь от этого маленького, но замечательного примера, то обнаружите, что он – лишь вершина айсберга. Замыкания являются одной из уникальных черт Лиспа. Они открывают путь к новым возможностям в программировании, не достижимым в других языках.°

6.6. Пример: строители функций

Язык Dylan известен как гибрид Scheme и Common Lisp, использующий синтаксис Pascal.° Он имеет большой набор функций, которые возвращают функции. Помимо complement, рассмотренной нами в предыдущем разделе, Dylan включает compose, disjoin, conjoin, curry, rcurry и always. На рис. 6.2 приведены реализации этих функций в Common Lisp, а на рис. 6.3 показаны эквиваленты, следующие из этих определений.

Первая из них, compose, принимает одну или несколько функций и возвращает новую функцию, которая применяет их по очереди. Таким образом,

```
(compose #'a #'b #'c)
```

```
(defun compose (&rest fns)
  (destructuring-bind (fn1 . rest) (reverse fns)
    #'(lambda (&rest args)
        (reduce #'(lambda (v f) (funcall f v))
                rest
                :initial-value (apply fn1 args)))))

(defun disjoin (fn &rest fns)
  (if (null fns)
      fn
      (let ((disj (apply #'disjoin fns)))
        #'(lambda (&rest args)
            (or (apply fn args) (apply disj args))))))

(defun conjoin (fn &rest fns)
  (if (null fns)
      fn
      (let ((conj (apply #'conjoin fns)))
        #'(lambda (&rest args)
            (and (apply fn args) (apply conj args))))))

(defun curry (fn &rest args)
  #'(lambda (&rest args2)
      (apply fn (append args args2))))

(defun rcurry (fn &rest args)
  #'(lambda (&rest args2)
      (apply fn (append args2 args))))

(defun always (x) #'(lambda (&rest args) x))
```

Рис. 6.2. *Компоновщики функций из Dylan*

```
      cddr = (compose #'cdr #'cdr)
       nth = (compose #'car #'nthcdr)
      atom = (compose #'not #'consp)
           = (rcurry #'typep 'atom)
        <= = (disjoin #'< #'=)
     listp = (disjoin #'null #'consp)
           = (rcurry #'typep 'list)
        1+ = (curry #'+ 1)
           = (rcurry #'+ 1)
        1- = (rcurry #'- 1)
    mapcan = (compose (curry #'apply #'nconc) #'mapcar)
complement = (curry #'compose #'not)
```

Рис. 6.3. *Некоторые эквиваленты*

возвращает функцию, эквивалентную вызову:

```
#'(lambda (&rest args) (a (b (apply #'c args))))
```

Из приведенного примера следует, что последняя функция может принимать любое количество аргументов, в то время как остальные должны принимать ровно один.

Давайте создадим функцию, вычисляющую квадратный корень числа, округляющую результат и возвращающую его, заключенным в список:

```
> (mapcar (compose #'list #'round #'sqrt)
          '(4 9 16 25))
((2) (3) (4) (5))
```

Следующие две функции, disjoin и conjoin, принимают некоторый набор предикатов. Вызов disjoin возвращает предикат, возвращающий истину, если хотя бы один из заданных предикатов является истинным. Функция conjoin возвращает предикат, который будет истинным лишь в случае соответствия проверяемого аргумента всем заданным предикатам.

```
> (mapcar (disjoin #'integerp #'symbolp)
          '(a "a" 2 3))
(T NIL T  T)
> (mapcar (conjoin #'integerp #'oddp)
          '(a "a" 2 3))
(NIL NIL NIL T)
```

Рассматривая предикаты как множества, можно сказать, что disjoin возвращает объединение множеств, а conjoin – их пересечение.

Функции curry и rcurry («right curry») имеют общую идею с определенной в предыдущем разделе make-adder. Они обе принимают функцию и некоторые ее аргументы и возвращают функцию, которая ожидает получить лишь остающиеся аргументы. Каждое из следующих выражений равносильно (make-adder 3):

```
(curry #'+ 3)
(rcurry #'+ 3)
```

Разница между curry и rcurry станет заметна на примере функции, различающей свои аргументы. Если мы применим curry к –, то полученная функция, будет вычитать свой аргумент из заданного числа:

```
> (funcall (curry #'- 3) 2)
1
```

в то время как для rcurry полученная функция будет вычитать заданное число из своего аргумента:

```
> (funcall (rcurry #'- 3) 2)
-1
```

Наконец, always имеет свой аналог в Common Lisp – функцию constanly. Она принимает любой аргумент и возвращает функцию, всегда возвра-

щающую этот аргумент. Как и identity, она может использоваться в тех случаях, когда требуется передать аргумент-функцию.

6.7. Динамический диапазон

В разделе 2.11 было показано различие между локальными и глобальными переменными. В действительности, различают лексические переменные, которые имеют лексический диапазон, и специальные переменные, имеющие динамический диапазон.[1] На практике локальные переменные почти всегда являются лексическими, а глобальные переменные – специальными, поэтому обе пары терминов можно считать взаимозаменяемыми.

Внутри лексического диапазона символ будет ссылаться на переменную, которая имеет такое же имя в контексте, где появляется этот символ. Локальные переменные по умолчанию имеют лексическое окружение. Итак, если мы определим функцию в окружении, содержащем переменную x:

```
(let ((x 10))
  (defun foo ()
    x))
```

то символ x, используемый в функции, будет ссылаться на эту переменную независимо от того, была ли в окружении, из которого функция вызывалась, переменная x, имеющая другое значение:

```
> (let ((x 20)) (foo))
10
```

В динамическом же диапазоне используется то значение переменной, которое имеется в окружении, где вызывается функция, а не в окружении, где она была определена.° Чтобы заставить переменную иметь динамический диапазон, нужно объявить ее как special в том контексте, где она встречается. Если мы определим foo следующим образом:

```
(let ((x 10))
  (defun foo ()
    (declare (special x))
    x))
```

то переменная x больше не будет ссылаться на лексическую переменную, существующую в контексте, в котором функция была определена, а будет ссылаться на любую динамическую переменную x, которая существует в момент вызова функции:

```
> (let ((x 20))
    (declare (special x))
```

[1] Под диапазоном (scope) следует понимать область видимости; соответственно лексическая область видимости и динамическая область видимости. – *Прим. науч. ред.*

```
    (foo))
20
```

Форма declare может начинать любое тело кода, в котором создаются новые переменные. Декларация special уникальна тем, что может изменить поведение программы. В главе 13 будут рассмотрены другие декларации. Прочие из них являются всего лишь советами компилятору; они могут сделать программу быстрее, но не меняют ее поведение.

Глобальные переменные, установленные с помощью setf в toplevel, подразумеваются специальными:

```
> (setf x 30)
30
> (foo)
30
```

Но в исходном коде лучше не полагаться на такие неявные определения и использовать defparameter.

В каких случаях может быть полезен динамический диапазон? Обычно он применяется, чтобы присвоить некоторой глобальной переменной новое временное значение. Например, для управления параметрами печати объектов используются 11 глобальных переменных, включая *print-base*, которая по умолчанию установлена как 10. Чтобы печатать числа в шестнадцатеричном виде (с основанием 16), можно привязать *print-base* к соответствующей базе:

```
> (let ((*print-base* 16))
    (princ 32))
20
32
```

Здесь отображены два числа: вывод princ и значение, которое она возвращает. Они представляют собой одно и то же значение, напечатанное сначала в шестнадцатеричном формате, так как *print-base* имела значение 16, а затем в десятичном, поскольку на возвращаемое из let значение уже не действовало *print-base* 16.

6.8. Компиляция

В Common Lisp можно компилировать функции по отдельности или же файл целиком. Если вы просто наберете определение функции в toplevel:

```
> (defun foo (x) (+ x 1))
FOO
```

то многие реализации создадут интерпретируемый код. Проверить, является ли функция скомпилированной, можно с помощью compiled-function-p:

```
> (compiled-function-p #'foo)
NIL
```

Функцию можно скомпилировать, сообщив compile имя функции:

```
> (compile 'foo)
FOO
```

После этого интерпретированное определение функции будет заменено скомпилированным. Скомпилированные и интерпретированные функции ведут себя абсолютно одинаково, за исключением отношения к compiled-function-p.

Функция compile также принимает списки. Такое использование compile обсуждается на стр. 171.

К некоторым функциям compile не применима: это функции типа stamp или reset, определенные через toplevel в собственном (созданном let) лексическом контексте[1]. Вам придется набрать эти функции в файле, затем его скомпилировать и загрузить. Этот запрет установлен по техническим причинам, а не потому, что что-то не так с определением функций в иных лексических окружениях.

Чаще всего функции компилируются не по отдельности, а в составе файла с помощью compile-file. Эта функция создает скомпилированную версию заданного файла, как правило, с тем же именем, но другим расширением. После загрузки скомпилированного файла compiled-function-p вернет истину для любой функции из этого файла.

Если одна функция используется внутри другой, то она также должна быть скомпилирована. Таким образом, make-adder (стр. 119), будучи скомпилированной, возвращает скомпилированную функцию:

```
> (compile 'make-adder)
MAKE-ADDER
> (compiled-funcion-p (make-adder 2))
T
```

6.9. Использование рекурсии

В Лиспе рекурсия имеет большее значение, чем в других языках. Этому есть три основные причины:

1. *Функциональное программирование*. Рекурсивные алгоритмы в меньшей мере нуждаются в использовании побочных эффектов.
2. *Рекурсивные структуры данных*. Неявное использование указателей в Лиспе облегчает рекурсивное создание структур данных. Наиболее общей структурой такого типа является список: либо nil, либо cons, чей cdr – также список.
3. *Элегантность*. Лисп-программисты придают огромное значение красоте их программ, а рекурсивные алгоритмы зачастую выглядят намного элегантнее их итеративных аналогов.

---

[1] В Лиспах, существовавших до ANSI Common Lisp, первый аргумент compile не мог быть уже скомпилированной функцией.

Поначалу новичкам бывает сложно понять рекурсию. Но, как было показано в разделе 3.9, вам вовсе не обязательно представлять себе всю последовательность вызовов рекурсивной функции, чтобы проверить ее корректность.

Это же утверждение верно и в том случае, когда вам необходимо написать свою рекурсивную функцию. Если вы сможете сформулировать рекурсивное решение проблемы, то вам не составит труда перевести это решение в код. Чтобы решить задачу с помощью рекурсии, необходимо сделать следующее:

1. Нужно показать, как можно решить ее с помощью разделения на конечное число похожих, но более мелких подзадач.
2. Нужно показать, как решить самую маленькую подзадачу (базовый случай) с помощью конечного набора операций.

Если вы в состоянии сделать это, значит, вы готовы к написанию рекурсивной функции. Теперь вам известно, что конечная проблема в конце концов будет разрешена, если на каждом шаге рекурсии она уменьшается, а самый маленький вариант требует конечного числа шагов для решения.

Например, в предложенном ниже рекурсивном алгоритме нахождения длины списка мы на каждом шаге рекурсии находим длину уменьшенного списка:

1. В общем случае длина списка равна длине его `cdr`, увеличенной на `1`.
2. Длину пустого списка принимаем равной `0`.

Когда это определение переводится в код, сначала идет базовый случай. Однако этап формализации задачи обычно начинается с рассмотрения наиболее общего случая.

Рассмотренный выше алгоритм описывает нахождение длины правильного списка. Определяя рекурсивную функцию, вы должны быть уверены, что разделение задачи действительно приводит к подзадачам меньшего размера. Переход к `cdr` списка приводит к меньшей задаче, однако лишь для списка, не являющегося циклическим.

Сейчас мы приведем еще два примера рекурсивных алгоритмов. Опять же они подразумевают конечный размер аргументов. Обратите внимание, что во втором алгоритме на каждом шаге рекурсии мы разбиваем задачу на *две* подзадачи.

`member` Объект содержится в списке, если он является его первым элементом или содержится в `cdr` этого списка. В пустом списке не содержится ничего.

`copy-tree` Копия дерева, представленного как `cons`-ячейка, – это ячейка, построенная из `copy-tree` для `car` исходной ячейки и `copy-tree` для ее `cdr`. Для атома `copy-tree` – это сам этот атом.

Сумев описать рекурсивный алгоритм таким образом, вы легко сможете написать соответствующее рекурсивное определение вашей функции.

Некоторые алгоритмы естественным образом ложатся на такие определения, но не все. Вам придется согнуться в три погибели, чтобы определить `our-copy-tree` (стр. 205) без использования рекурсии. С другой стороны, итеративный вариант `show-squares` (стр. 40) более доступен для понимания, нежели его рекурсивный аналог (стр. 41). Часто оптимальный выбор остается неясен до тех пор, пока вы не приступите к написанию кода.

Если производительность функции имеет для вас существенное значение, то следует учитывать два момента. Один из них – хвостовая рекурсия, которая будет обсуждаться в разделе 13.2. С хорошим компилятором не должно быть практически никакой разницы в скорости работы хвостовой рекурсии и цикла.[1] Однако иногда может оказаться проще переделать функцию в итеративную, чем модифицировать ее так, чтобы она удовлетворяла условию хвостовой рекурсивности.

Кроме того, вам необходимо помнить, что рекурсивный по сути алгоритм не всегда эффективен сам по себе. Классический пример – функция Фибоначчи. Эта функция рекурсивна по определению:

1. $\mathrm{Fib}(0) = \mathrm{Fib}(1) = 1$.
2. $\mathrm{Fib}(n) = \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2)$.

При этом дословная трансляция данного определения в код:

```
(defun fib (n)
  (if (<= n 1)
      1
      (+ (fib (- n 1))
         (fib (- n 2)))))
```

дает совершенно неэффективную функцию. Дело в том, что такая функция вычисляет одни и те же вещи по несколько раз. Например, вычисляя `(fib 10)`, она вызовет `(fib 9)` и `(fib 8)`. Однако вычисление `(fib 9)` уже включает в себя `(fib 8)`, и получается, что она выполняет это вычисление заново.

Ниже приведена аналогичная итеративная функция:

```
(defun fib (n)
  (do ((i n (- i 1))
       (f1 1 (+ f1 f2))
       (f2 1 f1))
      ((<= i 1) f1)))
```

[1] В действительности, хвостовая рекурсия просто преобразуется в соответствующий цикл. Такая оптимизация хвостовой рекурсии входит в стандарт языка Scheme, но отсутствует в Common Lisp. Тем не менее многие компиляторы Common Lisp поддерживают такую оптимизацию. – *Прим. перев.*

Итеративная версия менее понятна, зато намного более эффективна. Как часто на практике имеют место такие различия? Очень редко, и именно поэтому в различных книгах приводится один и тот же пример. Тем не менее таких ситуаций следует остерегаться.

Итоги главы

1. Именованная функция хранится как `symbol-function` соответствующего символа. Макрос `defun` скрывает подобные детали. Кроме того, он позволяет сопровождать функции документацией, а также настраивать поведение `setf`.
2. Имеется возможность определять локальные функции; этот процесс напоминает определение локальных переменных.
3. Функции могут иметь необязательные и остаточные аргументы, а также аргументы по ключу.
4. Утилиты дополняют язык. Они представляют собой пример программирования снизу-вверх в маленьком масштабе.
5. Лексические переменные существуют до тех пор, пока на них ссылается какой-либо объект. Замыкания – это функции, ссылающиеся на свободные переменные. Вы можете самостоятельно определять функции, возвращающие замыкания.
6. Dylan предоставляет функции для построения других функций. С использованием замыканий мы легко можем реализовать их в Common Lisp.
7. Специальные переменные имеют динамический диапазон.
8. Функции могут компилироваться индивидуально или (чаще) в составе файла.
9. Рекурсивный алгоритм решает задачу разделением ее на конечное число похожих подзадач меньшего размера.

Упражнения

1. Определите версию `tokens` (стр. 81), которая использует ключи `:test` и `:start`, по умолчанию равные `#'constituent` и `0` соответственно.
2. Определите версию `bin-search` (стр. 75), использующую ключи `:key`, `:test`, `:start` и `:end`, имеющие обычные для них значения по умолчанию.
3. Определите функцию, принимающую любое количество аргументов и возвращающую их количество.
4. Измените функцию `most` (стр. 118) так, чтобы она возвращала два значения – два элемента, имеющие наибольший вес.
5. Определите `remove-if` (без аргументов по ключу) с помощью `filter` (стр. 117).

6. Определите функцию, принимающую одно число и возвращающую наибольшее число из всех ранее полученных ею.
7. Определите функцию, принимающую одно число и возвращающую его же, если оно больше числа, переданного этой функции на предыдущем вызове. Во время первого вызова эта функция должна возвращать `nil`.
8. Предположим, что функция `expensive` имеет один аргумент – целое число между 0 и 100 включительно. Она производит сложные и дорогостоящие вычисления. Определите функцию `frugal`, возвращающую тот же ответ, что и `expensive`, но вызывающую `expensive`, лишь когда ей передается аргумент, который не встречался ранее.
9. Определите функцию наподобие `apply`, но где все числа, которые могут быть напечатаны при ее выполнении, выводятся в восьмеричном формате `(base 8)`.

7

Ввод и вывод

Common Lisp имеет богатые возможности для осуществления ввода-вывода. Для ввода, наряду с обычными инструментами для чтения символов, у нас есть `read`, который включает в себя полноценный парсер. Для вывода, вместе с привычными средствами для записи символов, мы получаем `format`, который сам по себе является небольшим языком. В этой главе вводятся все основные понятия ввода-вывода.

Существует два вида потоков: потоки знаков и бинарные потоки. В этой главе рассмотрены лишь потоки знаков; бинарные потоки будут описаны в разделе 14.2.

7.1. Потоки

Потоки – это объекты Лиспа, представляющие собой источники и/или приемники для передачи символов. Чтобы прочитать или записать файл, необходимо открыть соответствующий поток. Однако поток – это не то же самое, что файл. Когда вы читаете или печатаете что-либо в toplevel, вы также используете поток. Создавая потоки, вы даже можете читать из строк или писать в них.

По умолчанию для ввода используется поток `*standard-input*`, для вывода – `*standard-output*`. Первоначально они, как правило, ссылаются на один и тот же поток, соответствующий toplevel.

С функциями `read` и `format` мы уже знакомы. Ранее мы пользовались ими для чтения и печати в toplevel. Функция `read` имеет необязательный аргумент, который определяет входной поток и по умолчанию установлен в `*standard-input*`. Первый аргумент функции `format` также может быть потоком. Ранее мы вызывали `format` с первым аргументом `t`, который соответствует потоку `*standard-output*`. Таким образом, до этого момента мы наблюдали лишь поведение этих функций при использовании аргу-

ментов по умолчанию. Однако те же операции мы можем совершать и с любыми другими потоками.

Путь (*pathname*) – это переносимый способ определения пути к файлу. Путь имеет шесть компонентов: хост, устройство, каталог, имя, тип и версию. Вы можете создать путь с помощью `make-pathname` с одним или более аргументами по ключу. В простейшем случае достаточно задать лишь имя, оставив остальные параметры со значениями по умолчанию:

```
> (setf path (make-pathname :name "myfile"))
#P"myfile"
```

Базовая функция для открытия файлов – `open`. Ей необходимо сообщить путь[1], а ее поведением можно управлять с помощью многочисленных аргументов по ключу. В случае успеха она возвращает поток, указывающий на файл.

Вам нужно указать, как вы собираетесь использовать создаваемый поток. Параметр `:direction` определяет, будет ли производиться чтение (`:input`), запись (`:output`) или и то и другое одновременно (`:io`). Если поток используется для чтения, то параметр `:if-exists` определяет его поведение в случае, если файл уже существует. При этом обычно используется `:supersede` (перезаписать поверх). Итак, создадим поток, через который мы сможем записать что-либо в файл `"myfile"`:

```
> (setf str (open path :direction :output
                       :if-exists :supersede))
#<Stream C017E6>
```

Способ отображения потоков при печати зависит от используемой реализации Common Lisp.

Если теперь мы укажем поток `str` первым аргументом функции `format`, она будет печатать свой вывод в этот поток, а не в toplevel:

```
> (format str "Something~%")
NIL
```

Но если мы теперь посмотрим на содержимое файла, то можем и не найти в нем нашу строку. Некоторые реализации осуществляют вывод порциями, и ожидаемое содержимое файла может появиться лишь после закрытия потока:

```
> (close str)
NIL
```

Всегда закрывайте файлы по окончании их использования. Пока вы не сделаете этого, вы не можете быть уверены относительно их содержимого. Если мы теперь взглянем на содержимое файла `"myfile"`, то обнаружим строчку:

```
Something
```

[1] Вместо пути можно передать просто строку, однако такой способ не переносим между различными ОС.

Если мы хотим всего лишь прочитать содержимое файла, то откроем поток с аргументом `:direction :input`:

```
> (setf str (open path :direction :input))
#<Stream C01C86>
```

Содержимое файла можно читать с помощью любой функции чтения. Более подробно ввод описывается в разделе 7.2. Приведем пример, в котором для чтения строки из файла используется функция `read-line`:

```
> (read-line str)
"Something"
NIL
> (close str)
NIL
```

Не забудьте закрыть файл после завершения чтения.

Функции `open` и `close` практически никогда не используются явно. Гораздо удобнее использовать макрос `with-open-file`. Первый его аргумент – список, содержащий некоторое имя переменной и все те же аргументы, которые вы могли бы передать функции `open`. Далее следуют выражения, которые могут использовать заданную выше переменную, связанную внутри тела макроса с именем переменной. После выполнения всех выражений тела макроса поток закрывается автоматически. Таким образом, операция записи в файл может быть целиком выражена так:

```
(with-open-file (str path :direction :output
                          :if-exists :supersede)
  (format str "Something~%"))
```

Макрос `with-open-file` использует `unwind-protect` (стр. 295), чтобы гарантировать, что файл будет действительно закрыт, даже если выполнение выражений внутри тела макроса будет аварийно прервано.

7.2. Ввод

Двумя наиболее популярными функциями чтения являются `read-line` и `read`. Первая читает все знаки до начала новой строки, возвращая их в виде строки. Один ее необязательный параметр, как и для `read`, определяет входной поток. Если он не задан явно, то по умолчанию используется `*standard-input*`:

```
> (progn
    (format t "Please enter your name: ")
    (read-line))
Please enter your name: Rodrigo de Bivar
"Rodrigo de Bivar"
NIL
```

Эту функцию стоит использовать, если вы хотите получить значения такими, какими они были введены. (Вторым возвращаемым значением

будет t, если read-line закончит чтение файла прежде, чем встретит конец строки.)

В общем случае read-line принимает четыре необязательных аргумента: поток; параметр, уведомляющий, вызывать ли ошибку по достижении конца файла (eof); параметр, обозначающий, что возвращать, если не вызывать ошибку. Четвертый аргумент (стр. 242) обычно можно проигнорировать.

Итак, отобразить содержимое файла в toplevel можно с помощью следующей функции:

```
(defun pseudo-cat (file)
  (with-open-file (str file :direction :input)
    (do ((line (read-line str nil 'eof)
               (read-line str nil 'eof)))
        ((eql line 'eof))
      (format t "~A~%" line))))
```

Если вы хотите, чтобы ввод проходил синтаксический разбор как объекты Лиспа, используйте read. Эта функция считывает ровно одно выражение и останавливается в его конце. Это значит, что она может считывать больше строки или меньше строки. Разумеется, считываемый объект должен быть синтаксически корректным (valid) с точки зрения синтаксиса Лиспа.

Применяя функцию read в toplevel, мы сможем использовать сколько угодно переносов строк внутри одного выражения:

```
> (read)
(a
b
c)
(A B C)
```

С другой стороны, если на одной строке будет находиться несколько объектов Лиспа, read прекратит считывание знаков, закончив чтение первого объекта; оставшиеся знаки будут считываться при следующих вызовах read. Проверим это предположение с помощью ask-number (стр. 37). Введем одновременно несколько выражений и посмотрим, что происходит:

```
> (ask-number)
Please enter a number. a b
Please enter a number. Please enter a number. 43
43
```

Два приглашения напечатаны одно за другим на одной строке. Первый вызов read возвращает символ a, не являющийся числом, поэтому функция заново предлагает ввести число. Первый вызов считал только a и остановился. Поэтому следующий вызов read сразу считывает объект b, который также не число, и приглашение выводится еще раз.

Чтобы избежать подобных случаев, не стоит применять read напрямую. Лучший способ: пользовательский ввод читается с помощью read-line и далее обрабатывается с помощью read-from-string.° Эта функция принимает строку и возвращает первый объект, прочитанный из нее:

```
> (read-from-string "a b c")
A
2
```

Помимо прочитанного объекта она возвращает позицию в строке, на которой завершилось чтение.

В общем случае read-from-string может принимать два необязательных аргумента и три аргумента по ключу. Два необязательных – те же, что и третий и четвертый аргументы в read-line (вызывать ли ошибку при достижении конца строки и, если нет, что возвращать в этом случае). А аргументы по ключу :start и :end ограничивают часть строки, в которой будет выполняться чтение.

Рассмотренные в этом разделе функции определены с помощью более примитивной read-char, считывающей одиночный знак. Она принимает те же четыре необязательных аргумента, что и read, и read-line. Common Lisp также определяет функцию peek-char, которая похожа на read-char, но не удаляет прочитанный знак из потока.

7.3. Вывод

Имеются три простейшие функции вывода: prin1, princ и terpri. Всем им можно сообщить выходной поток; по умолчанию это *standard-output*.

Разница между prin1 и princ, грубо говоря, в том, что prin1 генерирует вывод для программ, а princ – для людей. Так, например, prin1 печатает двойные кавычки вокруг строк, а princ – нет:

```
> (prin1 "Hello")
"Hello"
"Hello"
> (princ "Hello")
Hello
"Hello"
```

Обе функции возвращают свой первый аргумент, который, кстати, совпадает с тем, который отображает prin1. Функция terpri печатает только новую строку.

Зная об этих функциях, вам легче будет понять поведение более общей format. С помощью функции format может быть осуществлен практически любой вывод. Она принимает поток (а также t или nil), строку форматирования и ноль или более аргументов. Строка форматирования может содержать *директивы форматирования*, которые начинаются со знака ~ (тильда). На их место будут выведены представления аргументов, переданных format.

Передавая t в качестве первого аргумента, мы направляем вывод в *standard-output*. Если первый аргумент – nil, то format возвратит строку вместо того, чтобы ее напечатать. Ради краткости будем приводить только такие примеры. Функцию format можно рассматривать одновременно и как невероятно мощный, и как жутко сложный инструмент. Она понимает огромное количество директив, но только часть из них вам придется использовать. Две наиболее распространенные директивы: ~A и ~%. (Между ~a и ~A нет различия, однако принято использовать вторую директиву, потому что в строке форматирования она более заметна.) ~A – это место для вставки значения, которое будет напечатано с помощью princ. ~% соответствует новой строке.

```
> (format nil "Dear ~A,~% Our records indicate…"
          "Mr. Malatesta")
"Dear Mr. Malatesta,
 Our records indicate…"
```

В этом примере format возвращает один аргумент – строку, содержащую символ переноса строки.

Директива ~S похожа на ~A, однако она выводит объекты так же, как prin1, а не как princ:

```
> (format t "~S ~A" "z" "z")
"z" z
NIL
```

Директивы форматирования сами могут принимать аргументы. ~F используется для печати выровненных справа чисел с плавающей запятой[1] и может принимать пять аргументов:

1. Суммарное количество выводимых символов. По умолчанию это длина числа.
2. Количество выводимых символов после точки. По умолчанию выводятся все.
3. Смещение точки влево (смещение на один знак соответствует умножению на 10). По умолчанию отсутствует.
4. Символ, который будет выведен вместо числа, если оно не умещается в количество символов, разрешенное первым аргументом. Если ничего не задано, то число, превышающее допустимый лимит, будет напечатано как есть.
5. Символ, печатаемый перед левой цифрой. По умолчанию – пробел.

[1] В англоязычной литературе принято использовать точку в качестве разделителя в десятичных дробях. В русском языке принято использовать запятую, а такие числа называть «числами с плавающей запятой» вместо «floating point numbers» (числа с плавающей точкой). В Common Lisp для записи десятичных дробей всегда используется точка. – *Прим. перев.*

Вот пример, в котором используются все пять аргументов:

```
> (format nil "~10,2,0,'*,' F" 26.21875)
"     26.22"
```

Исходное число округляется до двух знаков после точки (сама точка смещается на 0 положений влево, то есть не смещается), для печати числа выделяется пространство в 10 символов, на месте незаполненных слева полей печатаются пробелы. Обратите внимание, что символ звездочки передается как `'*`, а не как обычно `#\*`. Поскольку заданное число вписывается в предложенное пространство 10 символов, четвертый аргумент не используется.

Все эти аргументы не обязательны для использования. Чтобы применить значение по умолчанию, достаточно просто пропустить соответствующий аргумент. При желании напечатать число, округленное до двух знаков после точки, достаточно написать:

```
> (format nil "~,2,,,F" 26.21875)
"26.22"
```

Такая последовательность запятых может быть опущена, поэтому более распространена следующая запись:

```
> (format nil "~,2F" 26.21875)
"26.22"
```

Предупреждение: Округляя числа, `format` не гарантирует округление в большую или меньшую сторону. Поэтому `(format nil "~,1F" 1.25)` может выдать либо `"1.2"`, либо `"1.3"`. Таким образом, если вам нужно округление в конкретную сторону (например, при конвертации валют), перед печатью округляйте выводимое число явным образом.

7.4. Пример: замена строк

В этом разделе приведен пример использования ввода-вывода – простая программа для замены строк в текстовых файлах. Мы создадим функцию, которая сможет заменить каждую строку `old` в файле на строку `new`. Простейший способ сделать это – сверять каждый символ с первым символом `old`. Если они не совпадают, то мы можем просто напечатать символ из файла. Если они совпадают, то сверяем следующий символ из файла со *вторым* символом `old`, и т. д. Если найдено совпадение, то печатаем на выход строку `new`.°

Что происходит, когда мы натыкаемся на несовпадение в процессе сверки символов? Например, предположим, что мы ищем шаблон `"abac"`, а входной файл содержит `"ababac"`. Совпадение будет наблюдаться вплоть до четвертого символа: в файле это `b`, а в шаблоне `c`. Дойдя до четвертого символа, мы понимаем, что можем напечатать первый символ `a`. Однако некоторые пройденные символы нам все еще нужны, потому что третий прочтенный символ `a` совпадает с первым символом шаблона. Таким

образом, нам понадобится место, где мы будем хранить все прочитанные символы, которые нам пока еще нужны.

Очередь для временного хранения входной информации называется *буфером*. В данном случае необходимый размер буфера нам заранее неизвестен, и мы воспользуемся структурой данных под названием *кольцевой буфер*. Кольцевой буфер строится на основе вектора. Способ его использования напоминает кольцо: этот вектор последовательно наполняется вновь поступающими символами, а когда он заполнится полностью, процесс начинается с начала, поверх уже существующих элементов. Если нам заранее известно, что не понадобится хранить более n элементов, вектора длиной n будет достаточно, и перезапись элементов с начала не приведет к потере данных.

На рис. 7.1 показан код, реализующий операции с кольцевым буфером. Структура `buf` имеет пять полей: вектор для хранения объектов и четыре индекса. Два из них, `start` и `end`, необходимы для любых операций с кольцевым буфером: `start` указывает на начальное значение в буфере и будет увеличиваться при изъятии элемента из буфера; `end` указывает на последнее значение в буфере и будет увеличиваться при добавлении нового элемента.

Два других индекса, `used` и `new`, потребуются для использования буфера в нашем приложении. Они могут принимать значение между `start` и `end`, причем всегда будет соблюдаться следующее соотношение:

$$\texttt{start} \le \texttt{used} \le \texttt{new} \le \texttt{end}$$

Пару `used` и `new` можно считать аналогом `start` и `end` для текущего совпадения. Когда мы начинаем отслеживать совпадение, `used` будет равен `start`, а `new` — `end`. Для каждого последовательного совпадения пары символов `used` будет увеличиваться. Когда `used` достигнет `new`, это будет означать, что мы считали из буфера все элементы, занесенные туда до начала проверки данного совпадения. Нам не нужно использовать больше символов, чем было в буфере перед началом нахождения текущего совпадения, иначе мы бы использовали некоторые символы по нескольку раз. Поэтому нам требуется другой индекс, `new`, который исходно равен `end`, но не увеличивается при добавлении новых символов во время проверки совпадения.

Функция `bref` возвращает элемент, хранящийся в буфере по заданному индексу. Используя остаток от деления заданного индекса на длину вектора, мы можем сделать вид, что наш буфер имеет неограниченный размер. Вызов (`new-buf` n) создает новый буфер, способный хранить до n элементов.

Чтобы поместить в буфер новое значение, будем применять `buf-insert`. Эта функция увеличивает индекс `end` и кладет на это место заданный элемент. Обратной функцией является `buf-pop`, которая возвращает первый элемент буфера и увеличивает индекс `start`. Эти две функции нужны для любого кольцевого буфера.

```
(defstruct buf
  vec (start -1) (used -1) (new -1) (end -1))

(defun bref (buf n)
  (svref (buf-vec buf)
         (mod n (length (buf-vec buf)))))

(defun (setf bref) (val buf n)
  (setf (svref (buf-vec buf)
               (mod n (length (buf-vec buf))))
        val))

(defun new-buf (len)
  (make-buf :vec (make-array len)))

(defun buf-insert (x b)
  (setf (bref b (incf (buf-end b))) x))

(defun buf-pop (b)
  (prog1
    (bref b (incf (buf-start b)))
    (setf (buf-used b) (buf-start b)
          (buf-new  b) (buf-end   b))))

(defun buf-next (b)
  (when (< (buf-used b) (buf-new b))
    (bref b (incf (buf-used b)))))

(defun buf-reset (b)
  (setf (buf-used b) (buf-start b)
        (buf-new  b) (buf-end   b)))

(defun buf-clear (b)
  (setf (buf-start b) -1 (buf-used  b) -1
        (buf-new   b) -1 (buf-end   b) -1))

(defun buf-flush (b str)
  (do ((i (1+ (buf-used b)) (1+ i)))
      ((> i (buf-end b)))
    (princ (bref b i) str)))
```

Рис. 7.1. *Операции с кольцевым буфером*

Следующие две функции написаны специально для нашего приложения: `buf-next` читает значение из буфера без его извлечения, `buf-reset` сбрасывает `used` и `new` до исходных значений `start` и `end`. Если все значения до `new` прочитаны, `buf-next` возвратит `nil`. Поскольку мы собираемся хранить в буфере исключительно символы, отличить `nil` как особое значение не будет проблемой.

Наконец, `buf-flush` выводит содержимое буфера, записывая все доступные элементы в заданный поток, а `buf-clear` очищает буфер, сбрасывая все индексы до -1.

Функции из кода на рис. 7.1 используются в коде на рис. 7.2, предоставляющем средства для замены строк. Функция `file-subst` принимает четыре аргумента: искомую строку, ее замену, входной и выходной файлы. Она создает потоки, связанные с заданными файлами, и вызывает функцию `stream-subst`, которая и выполняет основную работу.

Вторая функция, `stream-subst`, использует алгоритм, который был схематически описан в начале раздела. Каждый раз она читает из входного потока один символ. Если он не совпадает с первым элементом искомой строки, он тут же записывается в выходной поток (1). Когда начинается совпадение, символы ставятся в очередь в буфер `buf` (2).

```
(defun file-subst (old new file1 file2)
  (with-open-file (in file1 :direction :input)
     (with-open-file (out file2 :direction :output
                                :if-exists :supersede)
       (stream-subst old new in out))))

(defun stream-subst (old new in out)
  (let* ((pos 0)
         (len (length old))
         (buf (new-buf len))
         (from-buf nil))
    (do ((c (read-char in nil :eof)
            (or (setf from-buf (buf-next buf))
                (read-char in nil :eof))))
        ((eql c :eof))
      (cond ((char= c (char old pos))
             (incf pos)
             (cond ((= pos len)            ; 3
                    (princ new out)
                    (setf pos 0)
                    (buf-clear buf))
                   ((not from-buf)         ; 2
                    (buf-insert c buf))))
            ((zerop pos)                   ; 1
             (princ c out)
             (when from-buf
               (buf-pop buf)
               (buf-reset buf)))
            (t                             ; 4
             (unless from-buf
               (buf-insert c buf))
             (princ (buf-pop buf) out)
             (buf-reset buf)
             (setf pos 0))))
    (buf-flush buf out)))
```

Рис. 7.2. *Замена строк*

Переменная pos указывает на положение символа в искомой строке. Если оно равно длине самой строки, это означает, что совпадение найдено. В таком случае в выходной поток записывается замена строки, а буфер очищается (3). Если в какой-либо момент последовательность символов перестает соответствовать искомой строке, мы вправе забрать первый символ из буфера и записать в выходной поток, а также установить pos равным нулю, а сам буфер сбросить (4).

Следующая таблица наглядно демонстрирует, что происходит при замене "baro" на "baric" в файле, содержащем только одно слово barbarous:

Символ	Источник	Совпадение	Вариант	Вывод	Буфер
b	файл	b	2		b
a	файл	a	2		b a
r	файл	r	2		b a r
b	файл	o	4	b	b.a r b.
a	буфер	b	1	a	a.r b.
r	буфер	b	1	r	r.b.
b	буфер	b	1		r b:
a	файл	a	2		r b:a
r	файл	r	2		r b:a r
o	файл	o	3	baric	
u	файл	b	1	u	
s	файл	b	1	s	

Первая колонка содержит текущий символ – значение переменной c; вторая показывает, откуда он был считан – из буфера или же прямиком из файла; третья показывает символ, с которым мы сравниваем, – элемент old по индексу pos; четвертая указывает на вариант действия, совершаемого в данном случае; пятая колонка соответствует текущему содержимому буфера после выполнения операции. В последней колонке точками после символа показаны также позиции used и new. Если они совпадают, то это отображается с помощью двоеточия.

Предположим, что у нас есть файл "test1", содержащий следующий текст:

```
The struggle between Liberty and Authority is the most conspicuous feature
in the portions of history with which we are earliest familiar, particularly
in that of Greece, Rome, and England.
```

После выполнения (file-subst " th" " z" "test1" "test2") файл "test2" будет иметь следующий вид:

```
The struggle between Liberty and Authority is ze most conspicuous feature in
ze portions of history with which we are earliest familiar, particularly in
zat of Greece, Rome, and England.
```

Чтобы максимально упростить пример, код на рис. 7.2. просто заменяет одну строку на другую. Однако его несложно обобщить и соорудить поиск по полноценным шаблонам вместо последовательностей букв. Все, что вам потребуется, – заменить вызов `char=` на функцию проверки на соответствие шаблону.

7.5. Макрознаки

Макрознаки (*macro character*) – это знаки, имеющие особое значение для функции `read`. Знаки `a` и `b` обычно обрабатываются как есть, однако знак открывающей круглой скобки распознается как начало считывания списка.

Макрознаки или их комбинация известны также как *макросы чтения* (*read-macro*). Многие макросы чтения в Common Lisp на деле являются сокращениями. Например, кавычка. Цитируемое выражение, например `'a`, при обработке с помощью `read` раскрывается в список `(quote a)`. При наборе подобного выражения в toplevel оно вычисляется сразу же после прочтения, и вы никогда не увидите этого преобразования. Его можно обнаружить, вызывая `read` явно:

```
> (car (read-from-string "'a"))
QUOTE
```

Макрос чтения, соответствующий `quote`, состоит лишь из одного знака, что является редкостью. С ограниченным набором знаков сложно получить много односимвольных макросов чтения. Большинство макросов чтения состоит из двух или более знаков.

Такие макросы чтения называются *диспетчеризуемыми* (*dispatching*), а их первый знак называется диспетчером. У всех стандартных макросов чтения знаком-диспетчером является решетка `#`. С некоторыми из них мы уже знакомы. Например, `#'` соответствует `(function ...)`, а `'` является сокращением для `(quote ...)`.

Следующие диспетчеризуемые макросы чтения мы также уже видели: `#(...)` является сокращением для векторов, `#nA(...)` – для массивов, `#\` – для знаков, `#S(n ...)` – для структур. При выводе на печать через `prin1` (или `format` с `~S`) такие объекты отобразятся в виде соответствующих макросов чтения.[1] Это означает, что объекты, записанные буквально, могут быть считаны обратно в таком же виде:

```
> (let ((*print-array* t))
    (vectorp (read-from-string (format nil "~S"
                                       (vector 1 2)))))
T
```

[1] Чтобы векторы и массивы печатались таким образом, необходимо установить значение `*print-array*`, равное `t`.

Разумеется, на выходе получается не тот же самый вектор, а другой, но состоящий из тех же элементов.

Не все объекты отображаются в соответствии с таблицей чтения `readtable`. Функции и хеш-таблицы, к примеру, отображаются через `#<...>`. Фактически `#<` – тоже макрос чтения, однако при передаче в `read` он всегда вызывает ошибку. Функции и хеш-таблицы не могут быть напечатаны, а затем прочитаны обратно, и этот макрос чтения гарантирует, что пользователи не будут питать иллюзий на этот счет[1].

При определении собственного представления для какого-либо объекта (например, для отображения структур) важно помнить этот принцип. Либо объект в вашем представлении может быть прочитан обратно, либо вы используете `#<...>`.

Итоги главы

1. Потоки – источники ввода и получатели вывода. В потоках знаков ввод и вывод состоят из знаков.
2. Поток, используемый по умолчанию, соответствует toplevel. При открытии файлов создаются новые потоки.
3. Ввод может обрабатываться как набор объектов, строка знаков или же как отдельные знаки.
4. Функция `format` предоставляет полное управление выводом.
5. Чтобы осуществить замену строк в текстовом файле, вам придется считывать знаки в буфер.
6. Когда `read` встречает макрознак типа `'`, она вызывает связанную с ним функцию.

Упражнения

1. Определите функцию, возвращающую список из строк, прочитанных из заданного файла.
2. Определите функцию, возвращающую список выражений, содержащихся в заданном файле.
3. Пусть в файле некоторого формата комментарии помечаются знаком `%`. Содержимое строки от начала знака комментария до ее конца игнорируется. Определите функцию, принимающую два имени файла и записывающую во второй содержимое первого с вырезанными комментариями.

---

[1] В Лиспе решетка с кавычкой не могут использоваться для описания функций, так как этот макрознак не в состоянии представлять замыкание.

4. Определите функцию, принимающую двумерный массив чисел с плавающей запятой и отображающую его в виде аккуратных колонок. Каждый элемент должен печататься с двумя знаками после запятой на пространстве 10 знаков. (Исходите из предположения, что все они уместятся в отведенном пространстве.) Вам потребуется функция `array-dimensions` (стр. 374).
5. Измените функцию `stream-subst` так, чтобы она понимала шаблоны с метазнаком `+`. Если знак `+` появляется в строке `old`, он может соответствовать любому знаку.
6. Измените функцию `stream-subst` так, чтобы шаблон мог содержать элемент, соответствующий: любой цифре, любой цифре и букве, любому знаку. Шаблон должен уметь распознавать любые читаемые знаки. (Подсказка: `old` теперь не обязательно должен быть строкой.)

8

Символы

С Лисп-символами вы уже довольно хорошо знакомы, но есть еще некоторые вещи, на которые стоит обратить внимание. Поначалу не стоило углубляться в механизм их реализации. Вы можете использовать их как в качестве объектов, так и в качестве имен для объектов, не задумываясь о том, каким образом связаны эти две роли. Но в какой-то момент полезно остановиться и разобраться с тем, что же происходит внутри. В этой главе подробно обсуждаются все детали, связанные с символами.

8.1. Имена символов

В главе 2 символы описывались как имена переменных, существующие сами по себе. Однако область применения символов в Лиспе шире, чем область применения переменных в большинстве других языков. На самом деле, имя символа – это обычная строка. Имя символа можно получить с помощью `symbol-name`:

```
> (symbol-name 'abc)
"ABC"
```

Обратите внимание, что имена символов записываются в верхнем регистре. По умолчанию Common Lisp не чувствителен к регистру, поэтому при чтении он преобразует все буквы в именах к заглавным:

```
> (eql 'aBc 'Abc)
T
> (CaR '(a b c))
A
```

Для работы с символами, названия которых содержат пробелы или другие знаки, влияющие на алгоритм считывания, используется особый синтаксис. Любая последовательность знаков между вертикальными

черточками, считается символом. Таким образом вы можете поместить любые знаки в имя символа:

```
> (list '|Lisp 1.5| '|| '|abc| '|ABC|)
(|Lisp 1.5| || |abc| ABC)
```

При считывании такого символа преобразование к верхнему регистру не производится, а макрознаки читаются как есть.

Так на какие же символы можно ссылаться, не прибегая к использованию вертикальной черты? По сути, на все символы, имена которых не являются числом и не содержат знаков, имеющих для `read` специального значения. Чтобы узнать, может ли символ существовать без такого оформления, достаточно просто посмотреть, как Лисп его напечатает. Если при печати символ не будет обрамлен вертикальными черточками, как последний символ в предыдущем примере, значит и вам не нужно их использовать.

Следует помнить, что вертикальные черточки – специальный синтаксис, а не часть имени символа:

```
> (symbol-name '|a b c|)
"a b c"
```

(Если вы желаете включить черточку в имя символа, расположите перед ней знак «\».)

8.2. Списки свойств

В Common Lisp каждый символ имеет собственный *список свойств* (*property-list*, *plist*). Функция `get` принимает символ и ключ, возвращая значение, связанное с этим ключом, из списка свойств:

```
> (get 'alizarin 'color)
NIL
```

Для сопоставления ключей используется `eql`. Если указанное свойство не задано, `get` возвращает `nil`.

Чтобы ассоциировать значение с ключом, можно использовать `setf` вместе с `get`:

```
> (setf (get 'alizarin 'color) 'red)
RED
> (get 'alizarin 'color)
RED
```

Теперь свойство `color` (цвет) символа `alizarin` имеет значение `red` (красный).

Функция `symbol-plist` возвращает список свойств символа:

```
> (setf (get 'alizarin 'transparency) 'high)
HIGH
> (symbol-plist 'alizarin)
(TRANSPARENCY HIGH COLOR RED)
```

Заметьте, что списки свойств не представляются как ассоциативные списки, хотя и работают похожим образом.

В Common Lisp списки свойств используются довольно редко. Они в значительной мере вытеснены хеш-таблицами.

8.3. А символы–то не маленькие

Символы создаются неявным образом, когда мы печатаем их имена, и при отображении самих символов их имена – это все что мы видим. При таких обстоятельствах легко подумать, что символ – это лишь имя и ничего более. Однако многое скрыто от наших глаз.

Из того, как мы видим и используем символы, может показаться, что это маленькие объекты, такие же как, скажем, целые числа. На самом деле, символы обладают внушительными размерами и более походят на структуры, определяемые через `defstruct`. Символ может иметь имя, пакет, значение связанной с ним переменной, значение связанной функции и список свойств. Устройство символа и взаимосвязь между его компонентами показаны на рис. 8.1.

Мало кто использует настолько большое количество символов, чтобы вставал вопрос об экономии памяти. Однако стоит держать в уме, что символы – полноценные объекты, а не просто имена. Две переменные могут ссылаться на один символ, так же как и на один список: в таком случае они имеют указатель на общий объект.

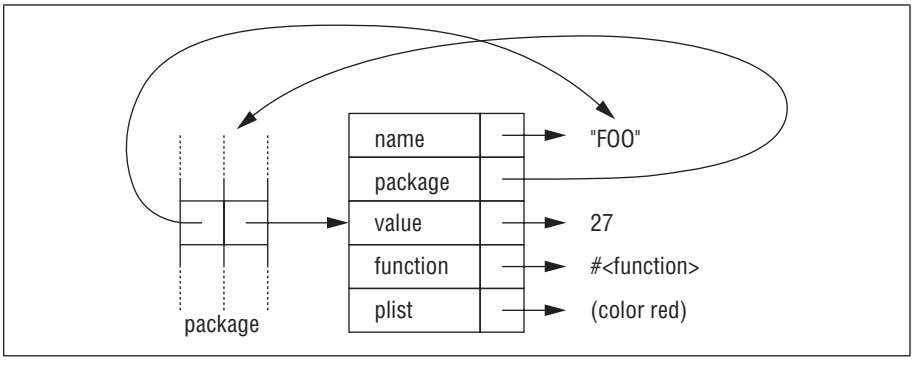

Рис. 8.1. *Структура символа*

8.4. Создание символов

В разделе 8.1 было показано, как получать имена символов. Также возможно и обратное действие – преобразование строки в символ. Это более сложная задача, потому что для ее выполнения необходимо иметь представление о пакетах.

Логически пакеты – это таблицы, отображающие имена в символы. Любой символ принадлежит конкретному пакету. Символ, принадлежащий пакету, называют *интернированным* в него. Пакеты делают возможной модульность, ограничивая область видимости символов. Имена функций и переменных считаются символами, поэтому они могут быть доступны в соответствующих пакетах.

Большинство символов интернируются во время считывания. Когда вы впервые вводите символ, Лисп создает новый символьный объект и интернирует его в текущий пакет (по умолчанию это `common-lisp-user`). Однако вы можете сделать то же самое вручную с помощью `intern`:

```
> (intern "RANDOM-SYMBOL")
RANDOM-SYMBOL
NIL
```

Функция `intern` принимает также дополнительный аргумент – пакет (по умолчанию используется текущий). Приведенный выше вызов создает в текущем пакете символ с именем `"RANDOM-SYMBOL"`, если такого символа пока еще нет в данном пакете. Второй аргумент возвращает истину, когда такой символ уже существует.

Не все символы являются интернированными. Иногда может оказаться полезным использование неинтернированных символов, так же как иногда лучше записывать номера телефонов не в записную книжку, а на отдельный клочок бумаги. Неинтернированные символы создаются с помощью `gensym`. Мы познакомимся с ними при разборе макросов в главе 10.

8.5. Использование нескольких пакетов

Большие программы часто разделяют на несколько пакетов. Если каждая часть программы расположена в собственном пакете, то разработчик другой ее части может смело использовать имена функций и переменных, имеющиеся в первой части.

При использовании языков, не поддерживающих разделение пространства имен, программистам, работающим над серьезными проектами, приходится вводить договоренности, чтобы избежать конфликтов при использования одних и тех же имен. Например, разработчик подсистемы отображения, вероятно, будет использовать имена, начинающиеся с `disp_`, в то время как программист, разрабатывающий математические процедуры, будет применять имена, начинающиеся с `math_`. Так, например, функция, выполняющая быстрое преобразование Фурье, вполне может носить имя `math_fft`.

Пакеты выполняют эту работу самостоятельно. Внутри отдельного пакета можно использовать любые имена. Только те символы, которые будут явным образом *экспортированы*, будут видимы в других пакетах, где будут доступны с помощью префикса (или *квалифицированы*), соответствующего имени содержащего их пакета.

Пусть, например, имеется программа, разделенная на два пакета, math и disp. Если символ fft экспортируется пакетом math, то в пакете disp он будет доступен как math:fft. Внутри пакета math к нему можно обращаться без префикса – просто fft.

Ниже приводится пример определения пакета, которое можно поместить в начало файла:

```
(defpackage "MY-APPLICATION"
            (:use "COMMON-LISP" "MY-UTILITIES")
            (:nicknames "APP")
            (:export "WIN" "LOSE" "DRAW"))

(in-package my-application)
```

Для создания нового пакета используется defpackage. Пакет my-application[1] *использует* два других пакета, common-lisp и my-utilities. Это означает, что символы, экспортируемые ими, доступны в новом пакете *без* использования префиксов. Практически всегда в создаваемых пакетах используется common-lisp, поскольку никто не хочет использовать квалифицированное имя с префиксом при каждом обращении к встроенным в Лисп операторам и переменным.

Пакет my-applications сам экспортирует лишь три символа: win, lose и draw. Поскольку в defpackage было также задано сокращенное имя (nickname) пакета – app, то и к экспортируемым символам можно будет обращаться также и через него: app:win.

За defpackage следует вызов in-package, которая выставляет текущим пакетом my-application. Теперь все новые символы, для которых не указан пакет, будут интернироваться в my-application до тех пор, пока in-package не изменит текущий пакет. После окончания загрузки файла, содержащего вызов in-package, значение текущего пакета сбрасывается в то, каким оно было до начала загрузки.

8.6. Ключевые слова

Символы пакета keyword (известные как *ключевые слова*) имеют две особенности: они всегда самовычисляемы и вы можете всегда ссылаться на них просто как на :x вместо keyword:x. Когда мы только начинали использовать аргументы по ключу (стр. 60), вызов (member '(a) '((a) (z)) test: #'equal) мог показаться нам более естественным, чем (member '(a) '((a) (z)) :test #'equal). Теперь мы понимаем, почему второе, слегка неестественное написание, является корректным. Двоеточие перед именем символа позволяет отнести его к ключевым словам.

[1] В этом примере используются имена, состоящие лишь из заглавных букв, потому что, как упомянуто в разделе 8.1, имена символов конвертируются в верхний регистр.

Зачем использовать ключевые слова вместо обычных символов? Потому что они доступны везде. Функция, принимающая символы в качестве аргументов, как правило, должна быть рассчитана на обработку именно ключевых слов. Например, следующий код может быть вызван из любого пакета без изменений:

```
(defun noise (animal)
  (case animal
    (:dog :woof)
    (:cat :meow)
    (:pig :oink)))
```

Если бы эта функция использовала обычные символы, то она могла бы вызываться лишь в исходном пакете до тех пор, пока не будут экспортированы все символы-ключи `case`-выражения.

8.7. Символы и переменные

В Лиспе есть определенная особенность, которая может смущать новичков: символы могут быть связаны с переменными двумя различными способами. Если символ является именем специальной переменной, ее значение хранится в одном из полей структуры символа (см. рис. 8.1). Получить доступ к этому полю можно с помощью `symbol-value`. Таким образом, существует прямая связь между символом и представляемой им специальной переменной.

С лексическими переменными дело обстоит иначе. Символ, используемый в качестве лексической переменной, всего лишь заменяет соответствующее ему значение. Компилятор будет заменять ссылку на лексическую переменную регистром или областью памяти. В полностью скомпилированном коде не будет каких-либо упоминаний об этом символе (разумеется, если не включена соответствующая опция отладки). Ни о какой связи между символом и значением лексической переменной говорить не приходится: к тому времени, как появляются значения, имена символов исчезают.

8.8. Пример: генерация случайного текста

Если вам предстоит написать программу, каким-либо образом работающую со словами, отличной идеей часто является использование символов вместо строк, потому что символы атомарны. Они могут сравниваться за одну итерацию с помощью `eql`, в то время как строки сравниваются побуквенно с помощью `string-equal` или `string=`. В качестве примера рассмотрим генерацию случайного текста. Первая часть программы будет считывать образец текста (чем он больше, тем лучше), собирая информацию о связях между соседними словами. Вторая ее часть будет случайным образом выполнять проходы по сети, построенной ранее из слов исходного текста. После прохождения каждого слова программа

будет делать взвешенный случайный шаг к следующему слову, встретившемуся после данного в оригинальном тексте. Полученный таким образом текст будет местами казаться довольно связным, потому что каждая пара слов в нем соотносится друг с другом. Поразительно, но целые предложения, а иногда даже и абзацы будут порой казаться вполне осмысленными.

На рис. 8.2 приводится первая часть программы – код для сбора информации из исходного текста. На его основе строится хеш-таблица `*words*`. Ключами в ней являются символы, соответствующие словам, а значениями будут ассоциативные списки следующего типа:

```
((|sin| . 1) (|wide| . 2) (|sights| . 1))
```

```
(defparameter *words* (make-hash-table :size 10000))

(defconstant maxword 100)

(defun read-text (pathname)
  (with-open-file (s pathname :direction :input)
    (let ((buffer (make-string maxword))
          (pos 0))
      (do ((c (read-char s nil :eof)
              (read-char s nil :eof)))
          ((eql c :eof))
        (if (or (alpha-char-p c) (char= c #\'))
            (progn
              (setf (aref buffer pos) c)
              (incf pos))
            (progn
              (unless (zerop pos)
                (see (intern (string-downcase
                               (subseq buffer 0 pos))))
                (setf pos 0))
              (let ((p (punc c)))
                (if p (see p)))))))))

(defun punc (c)
  (case c
    (#\. '|.|) (#\, '|,|) (#\; '|;|)
    (#\! '|!|) (#\? '|?|) ))

(let ((prev '|.|))
  (defun see (symb)
    (let ((pair (assoc symb (gethash prev *words*))))
      (if (null pair)
          (push (cons symb 1) (gethash prev *words*))
          (incf (cdr pair))))
    (setf prev symb)))
```

Рис. 8.2. *Чтение образца текста*

Выше приведено значение ключа `|discover|` для отрывка из «Paradise Lost» Мильтона[1]. Мы видим, что слово «discover» было использовано в поэме четыре раза: по одному разу перед словами «sin» и «sights» и дважды перед словом «wide». Подобная информация собирается функцией `read-text`, которая строит такой ассоциативный список для каждого слова в заданном тексте. При этом файл читается побуквенно, а слова собираются в строку `buffer`. С параметром `maxword=100` программа сможет читать слова, состоящие не более чем из ста букв. Для английских слов этого более чем достаточно.

Если считанный знак является буквой (это определяется с помощью `alpha-char-p`) или апострофом, то он записывается в буфер. Любой другой символ сигнализирует об окончании слова, после чего все накопленное слово, интернированное в символ, передается в функцию `see`. Некоторые знаки пунктуации также расцениваются как слова – функция `punc` выводит словесный эквивалент заданного знака пунктуации.

Функция `see` регистрирует каждое встреченное слово. Ей также необходимо иметь информацию о предыдущем слове, для чего используется переменная `prev`. Изначально она содержит псевдослово, соответствующее точке. Если `see` уже вызывалась хотя бы один раз, то `prev` содержит слово, переданное этой функции на предыдущем вызове.

После отработки `read-text` таблица `*words*` будет содержать вхождения всех слов в заданном тексте. Их количество можно подсчитать с помощью `hash-table-count`. Английские тексты, не отличающиеся особым словесным разнообразием, редко содержат более 10000 различных слов.

А вот теперь начинается самая веселая часть. На рис. 8.3 представлен код, генерирующий текст с помощью кода на рис. 8.2. Правит балом рекурсивная функция `generate-text`. Ей необходимо сообщить желаемое количество слов в создаваемом тексте и (необязательно) предыдущее слово. По умолчанию это точка, и текст начинается с нового предложения.

Для получения каждого последующего слова `generate-text` вызывает `random-text` с предыдущим словом. Функция `random-text` случайным образом выбирает одно из слов, следующих после `prev` в исходном тексте. Вероятность выбора одного из перечисленных слов определяется в соответствии с частотой его появления в тексте.°

Теперь самое время осуществить тестовый запуск нашей программы. Однако вы уже знакомы с примером того, что она может сделать: речь о той самой строфе в начале книги, для генерации которой был использован отрывок из «Paradise Lost» Мильтона.°

---

[1] Речь об эпической поэме английского мыслителя Джона Мильтона «Потерянный рай», изданной в 1667 году. Поэма написана белым стихом. – *Прим. перев.*

```
(defun generate-text (n &optional (prev '|.|))
  (if (zerop n)
      (terpri)
      (let ((next (random-next prev)))
        (format t "~A " next)
        (generate-text (1- n) next))))

(defun random-next (prev)
  (let* ((choices (gethash prev *words*))
         (i (random (reduce #'+ choices
                            :key #'cdr))))
    (dolist (pair choices)
      (if (minusp (decf i (cdr pair)))
          (return (car pair))))))
```

Рис. 8.3. *Генерация текста*

Итоги главы

1. Именем символа может быть любая строка, но символы, создаваемые с помощью `read`, по умолчанию преобразуются к верхнему регистру.
2. С символами связаны списки свойств, которые функционируют подобно ассоциативным спискам, хотя и имеют другую форму.
3. Символы – это довольно внушительные объекты и больше напоминают структуры, нежели просто имена.
4. Пакеты отображают строки в символы. Чтобы создать новый символ, принадлежащий пакету, необходимо этот символ интернировать в него. Символы не обязательно должны быть интернированы.
5. С помощью пакетов реализуется модульность, ограничивающая область действия имен. По умолчанию программы будут находиться в пользовательском пакете, однако большие программы часто разделяются на несколько пакетов, имеющих собственное назначение.
6. Символы одного пакета могут быть доступны в другом. Ключевые слова самовычисляемы и доступны из любого пакета.
7. В программах, работающих с отдельными словами, удобно представлять слова с помощью символов.

Упражнения

1. Могут ли два символа иметь одно имя, но не быть эквивалентными с точки зрения `eql`?
2. Оцените различие между количеством памяти, использованной для представления строки `"FOO"` и символа `foo`.

3. В вызове `defpackage`, приведенном на стр. 148, использовались лишь строки. Тем не менее вместо строк мы могли бы воспользоваться символами. Чем это может быть чревато?
4. Добавьте в программу на рис. 7.1 код, который помещает ее содержимое в пакет `"RING"`. Аналогично для кода на рис. 7.2 создайте пакет `"FILE"`. Уже имеющийся код должен остаться без изменений.
5. Напишите программу, проверяющую, была ли заданная цитата произведена с помощью Henley (см. раздел 8.8).
6. Напишите версию Henley, которая принимает слово и производит предложение, в середине которого находится заданное слово.

9

Числа

«Перемалывание чисел» (решение числовых задач большого объема) – одна из сильных сторон Лиспа. В нем имеется богатый набор числовых типов, а по их поддержке он даст фору многим другим языкам.

9.1. Типы

Common Lisp предоставляет множество различных числовых типов: целые числа, числа с плавающей запятой, рациональные и комплексные числа. Большинство функций, рассмотренных в этой главе, работают одинаково со всеми типами чисел, за исключением, может быть, комплексных чисел, к которым некоторые из этих функций не применимы.

Целое число записывается строкой цифр: `2001`. Число с плавающей запятой, помимо цифр, содержит десятичный разделитель – точку, например `253.72`, или `2.5372e2` в экспоненциальном представлении. Рациональная дробь представляется в виде отношения двух целых чисел: `2/3`. Комплексное число вида $a+bi$ можно записать как `#c(`a b`)`, где a и b – два действительных числа одного типа.

Проверку на принадлежность к соответствующему типу осуществляют предикаты `integerp`, `floatp` и `complexp`. Иерархия численных типов представлена на рис. 9.1.

Ниже приведены основные правила, согласно которым можно определить, число какого типа будет получено в результате вычисления:

1. Если функция принимает хотя бы одно число с плавающей запятой, она также вернет десятичную дробь (или комплексное число, компоненты которого – десятичные дроби). Так `(+ 1.0 2)` вернет `3.0`, а `(+ #c(0 1.0) 2)` вернет `#c(2.0 1.0)`.
2. Рациональная дробь при возможности будет сокращена до целого числа. Вызов `(/ 10 2)` вернет `5`.

3. Комплексные числа, мнимая часть которых равна нулю, будут преобразованы в действительные. Таким образом, вызов `(+ #c(1 -1) #c(2 1))` вернет `3`.

Правила 2 и 3 вступают в силу в момент считывания выражения, поэтому:

```
> (list (ratiop 2/2) (complexp #c(1 0)))
(NIL NIL)
```

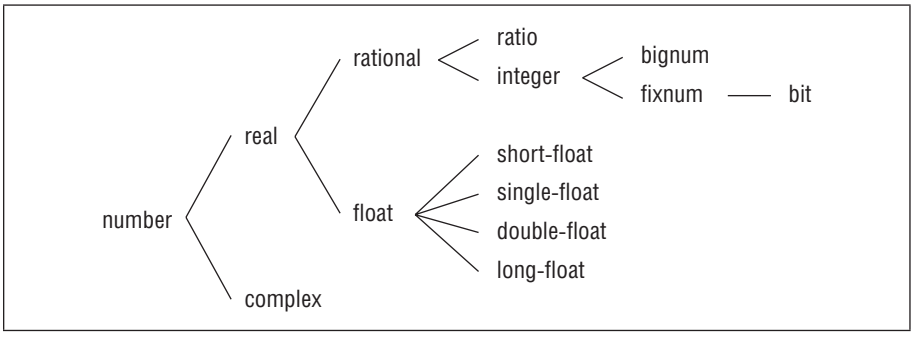

Рис. 9.1. *Численные типы*

9.2. Преобразование и извлечение

В Лиспе имеются средства для преобразования, а также извлечения компонентов чисел, принадлежащих любым типам. Функция `float` преобразует любое действительное число в эквивалентную ему десятичную дробь:

```
> (mapcar #'float '(1 2/3 .5))
(1.0 0.66666667 0.5)
```

Также можно конвертировать любое число в целое, но к этому *преобразованию* не всегда следует прибегать, поскольку это может повлечь за собой некоторую потерю информации. Целочисленную компоненту любого действительного числа можно получить с помощью функции `truncate`:

```
> (truncate 1.3)
1
0.2999995
```

Второе значение является результатом вычитания первого значения из аргумента. (Разница в `.00000005` возникает вследствие неизбежной неоднозначности операций с плавающей запятой.)

Функции `floor`, `ceiling` и `round` также приводят свои аргументы к целочисленным значениям. С помощью `floor`, возвращающей наибольшее целое число, меньшее или равное заданному аргументу, а также `ceiling`, возвращающей наименьшее целое, большее или равное аргументу, мы

можем обобщить функцию mirror? (стр. 62) для распознавания палиндромов:

```
(defun palindrome? (x)
  (let ((mid (/ (length x) 2)))
    (equal (subseq x 0 (floor mid))
           (reverse (subseq x (ceiling mid))))))
```

Как и truncate, функции floor и ceiling вторым значением возвращают разницу между аргументом и своим первым значением:

```
> (floor 1.5)
1
0.5
```

В действительности, мы могли бы определить truncate таким образом:

```
(defun our-truncate (n)
  (if (> n 0)
      (floor n)
      (ceiling n)))
```

Для получения ближайшего к аргументу целого числа существует функция round. В случае когда ее аргумент равноудален от обоих целых чисел, Common Lisp, как и многие другие языки, *не* округляет его, а возвращает ближайшее четное число:

```
> (mapcar #'round '(-2.5 -1.5 1.5 2.5))
(-2 -2 2 2)
```

Такой подход позволяет сгладить накопление ошибок округления в ряде приложений. Однако если необходимо округление вверх, можно написать такую функцию самостоятельно.[1] Как и остальные функции такого рода, round в качестве второго значения возвращает разницу между исходным и округленным числами.

Функция mod возвращает второе значение аналогичного вызова floor, а функция rem – второе значение вызова truncate. Мы уже использовали функцию mod (стр. 107) для определения делимости двух чисел и для нахождения позиции элемента в кольцевом буфере (стр. 137).

Для действительных чисел существует функция signum, которая возвратит 1, 0 или -1 в зависимости от того, каков знак аргумента: плюс, ноль или минус. Модуль числа можно получить с помощью функции abs. Таким образом, (* (abs *x*) (signum *x*)) = *x*.

```
> (mapcar #'signum '(-2 -0.0 0.0 0 .5 3))
(-1 -0.0 0.0 0 1.0 1)
```

В некоторых реализациях -0.0 может существовать сам по себе, как в примере, приведенном выше. В любом случае это не играет роли, и в вычислениях -0.0 ведет себя в точности как 0.0.

---

[1] Функция format при округлении не гарантирует даже того, будет ли получено четное или нечетное число. См. стр. 136.

Рациональные дроби и комплексные числа являются структурами, состоящими из двух частей. Получить соответствующие целочисленные компоненты рациональной дроби можно с помощью функций `numerator` и `denominator`. (Если аргумент уже является целым числом, то первая функция вернет сам аргумент, а последняя – единицу.) Аналогично функции `realpart` и `imagpart` извлекают действительную и мнимую части комплексного числа. (Если аргумент не комплексный, то первая вернет это число, а вторая – ноль.)

Функция `random` принимает целые числа или десятичные дроби. Выражение вида `(random` *n*`)` вернет число, большее или равное нулю и меньшее *n*, и это значение будет того же типа, что и аргумент.

9.3. Сравнение

Для сравнения двух чисел можно применять предикат `=`. Он возвращает истину, если аргументы численно эквивалентны, то есть их разность равна нулю:

```
> (= 1 1.0)
T
> (eql 1 1.0)
NIL
```

Он менее строг, чем `eql`, так как последний также требует, чтобы его аргументы были одного типа.

Предикаты для сравнения: `<` (меньше), `<=` (меньше или равно), `=` (равно), `>=` (больше или равно), `>` (больше) и `/=` (не равно). Все они принимают один или более аргументов. Вызванные с одним аргументом, они всегда возвращают истину. Для всех функций, кроме `/=`, вызов с тремя и более аргументами:

```
(<= w x y z)
```

равноценен объединению попарных сравнений:

```
(and (<= w x) (<= x y) (<= y z))
```

Так как `/=` возвращает истину, лишь когда *ни один* из аргументов не равен другому, выражение:

```
(/ w x y z)
```

эквивалентно

```
(and (/= w x) (/= w y) (/= w z)
     (/= x y) (/= x z) (/= y z))
```

Существуют также специализированные предикаты `zerop`, `plusp` и `minusp`, принимающие только один аргумент и возвращающие истину, если он `=`, `>` или `<` нуля соответственно. Эти предикаты взаимоисключающие. Что касается `-0.0` (если реализация его использует), то, несмотря на знак минус, это число `= 0`

```
> (list (minusp -0.0) (zerop -0.0))
(NIL T)
```

и соответствует zerop, а не minusp.

Предикаты oddp и evenp применимы лишь к целочисленным аргументам. Первый истинен для нечетных чисел, второй – для четных.

Из всех предикатов, упомянутых в этом разделе, лишь =, /= и zerop применимы к комплексным числам.

Наибольший и наименьший аргументы могут быть получены с помощью функций max и min соответственно, при этом должен быть задан хотя бы один аргумент:

```
> (list (max 1 2 3 4 5) (min 1 2 3 4 5))
(5 1)
```

Если среди аргументов есть хотя бы одно число с плавающей запятой, то тип возвращаемого значения зависит от используемой реализации.

9.4. Арифметика

Сложение и вычитание выполняются с помощью + и -. Обе функции могут работать с любым количеством аргументов, а в случае их отсутствия возвращают 0. Выражение вида (- *n*) возвращает –*n*. Выражение вида

```
(- x y z)
```

эквивалентно

```
(- (- x y) z)
```

Кроме того, имеются функции 1+ и 1-, которые возвращают свой аргумент, увеличенный или уменьшенный на 1 соответственно. Имя функции 1- может сбить с толку, потому что (1- *x*) возвращает *x* - 1, а не 1 - *x*.

Макросы incf и decf соответственно увеличивают и уменьшают свой аргумент. Выражение вида (incf *x n*) имеет такой же эффект, как и (setf *x* (+ *x n*)), а (decf *x n*) – как (setf *x* (- *x n*)). В обоих случаях второй аргумент не обязателен и по умолчанию равен 1.

Умножение выполняется с помощью функции *, которая принимает любое количество аргументов и возвращает их общее произведение. Будучи вызванной без аргументов, она возвращает 1.

Функция деления / требует задания хотя бы одного аргумента. Вызов (/ *n*) эквивалентен вызову (/ 1 *n*):

```
> (/ 3)
1/3
```

В то время как выражение:

```
(/ x y z)
```

эквивалентно

```
(/ (/ x y) z)
```

Обратите внимание на сходное поведение - и / в этом отношении.

Вызванная с двумя целыми аргументами, / возвращает рациональную дробь, если первый аргумент не делится на второй:

```
> (/ 365 12)
365/12
```

Если вы будете пытаться вычислять, к примеру, усредненную длину месяца, то вам может показаться, что toplevel над вами издевается. В подобных случаях, когда вам действительно нужна десятичная дробь, пользуйтесь функцией `float`:

```
> (float 365/12)
30.416666
```

9.5. Возведение в степень

Чтобы найти x^n, вызовем (`expt` x n):

```
> (expt 2 5)
32
```

А чтобы вычислить $log_n x$, вызовем (`log` x n):

```
> (log 32 2)
5.0
```

Обычно при вычислении логарифма возвращается число с плавающей запятой.

Для нахождения степени числа e существует специальная функция `exp`:

```
> (exp 2)
7.389056
```

Вычисление натурального логарифма выполняется функцией `log`, которой достаточно сообщить лишь само число:

```
> (log 7.389056)
2.0
```

Вычисление корней может выполняться с помощью `expt`, для чего ее второй аргумент должен быть рациональной дробью:

```
> (expt 27 1/3)
3.0
```

Но для вычисления квадратных корней быстрее работает функция `sqrt`:

```
> (sqrt 4)
2.0
```

9.6. Тригонометрические функции

Представлением числа π с плавающей запятой является константа `pi`. Ее точность зависит от используемой реализации. Функции `sin`, `cos` и `tan`

вычисляют соответственно синус, косинус и тангенс заданного в радианах угла:

```
> (let ((x (/ pi 4)))
    (list (sin x) (cos x) (tan x)))
(0.7071067811865475d0 0.707167811865476d0 1.0d0)
```

Все вышеперечисленные функции умеют работать с комплексными аргументами.

Обратные преобразования выполняются функциями `asin`, `acos`, `atan`. Для аргументов, находящихся на отрезке от -1 до 1, `asin` и `acos` возвращают действительные значения.

Гиперболические синус, косинус и тангенс вычисляются через `sinh`, `cosh` и `tanh` соответственно. Для них также определены обратные преобразования: `asinh`, `acosh`, `atanh`.

9.7. Представление

Common Lisp не накладывает каких-либо ограничений на размер целых чисел. Целые числа небольшого размера, которые помещаются в машинном слове, относятся к типу *fixnum*. Если для хранения числа требуется памяти больше, чем слово, Лисп переключается на представление *bignum*, для которого выделяется несколько слов для хранения целого числа. Это означает, что сам язык не накладывает каких-либо ограничений на размер чисел, и он зависит лишь от доступного объема памяти.

Константы `most-positive-fixnum` и `most-negative-fixnum` задают максимальные величины, которые могут обрабатываться реализацией без использования типа `bignum`. Во многих реализациях они имеют следующие значения:

```
> (values most-positive-fixnum most-negative-fixnum)
536870911
-536870912
```

Принадлежность к определенному типу проверяется с помощью предиката `typep`:

```
> (typep 1 'fixnum)
T
> (typep (1+ most-positive-fixnum) 'bignum)
T
```

В каждой реализации имеются свои ограничения на размер чисел с плавающей запятой. Common Lisp предоставляет четыре типа чисел с плавающей запятой: `short-float`, `single-float`, `double-float` и `long-float`. Реализации стандарта не обязаны использовать различные представления для разных типов (и лишь некоторые это делают).

Суть типа `short-float` в том, что он занимает одно машинное слово. Типы `single-float` и `double-float` занимают столько места, сколько нужно,

чтобы удовлетворять установленным требованиям к числам одинарной и двойной точности соответственно. Числа типа long-float могут быть большими настолько, насколько это требуется. Но в конкретной реализации все 4 типа могут иметь одинаковое внутреннее представление.

Тип числа можно задать принудительно, используя соответствующие буквы: s, f, d или l, а также e для экспоненциальной формы. (Заглавные буквы также допускаются, и этим стоит воспользоваться для представления типа long-float, потому что малую l легко спутать с цифрой 1.) Наибольшее представление числа 1.0 можно задать как 1L0.

Глобальные ограничения на размер чисел разных типов задаются шестнадцатью константами. Их имена выглядят как *m-s-f*, где *m* может быть most или least, *s* соответствует positive или negative, а *f* – один из четырех типов чисел с плавающей запятой.°

Превышение заданных ограничений приводит к возникновению ошибки в Common Lisp:

```
> (* most-positive-long-float 10)
Error: floating-point-overflow.
```

9.8. Пример: трассировка лучей

В качестве примера программы, построенной на численных расчетах, в этом разделе приводится решение задачи трассировки лучей. Трассировка лучей – это превосходный алгоритм рендеринга изображений, с помощью которого можно получать реалистичные картины. Однако данный метод является довольно дорогостоящим.

Для моделирования трехмерного изображения нам необходимо определить как минимум четыре предмета: наблюдателя, один или более источников света, набор объектов моделируемого мира и плоскость рисунка (*image plane*), которая служит окном в этот мир. Наша задача – сгенерировать изображение, соответствующее проекции мира на область плоскости рисунка.

Что же делает необычным метод трассировки лучей? Попиксельная отрисовка всей картины и симуляция прохождения луча через виртуальный мир. Такой подход позволяет достичь реалистичных оптических эффектов: прозрачности, отражений, затенений; он позволяет задавать отрисовываемый мир как набор геометрических тел, вместо того чтобы строить их из полигонов. Отсюда вытекает относительная прямолинейность в реализации метода.

На рис. 9.2 показаны основные математические утилиты, которыми мы будем пользоваться. Первая, sq, вычисляет квадрат аргумента. Вторая, mag, возвращает длину вектора по трем его компонентам x, y и z. Эта функция используется в следующих двух. unit-vector возвращает три значения, соответствующие координатам единичного вектора, имеющего то же направление, что и заданный:

```
> (multiple-value-call #'mag (unit-vector 23 12 47))
1.0
```

Кроме того, mag используется в функции distance, вычисляющей расстояние между двумя точками в трехмерном пространстве. (Определение структуры point содержит параметр :conc-name, равный nil. Это значит, что функции доступа к соответствующим полям структуры будут иметь такие же имена, как и сами поля, например x вместо point-x.)

```
(defun sq (x) (* x x))

(defun mag (x y z)
  (sqrt (+ (sq x) (sq y) (sq z))))

(defun unit-vector (x y z)
  (let ((d (mag x y z)))
    (values (/ x d) (/ y d) (/ z d))))

(defstruct (point (:conc-name nil))
  x y z)

(defun distance (p1 p2)
  (mag (- (x p1) (x p2))
       (- (y p1) (y p2))
       (- (z p1) (z p2))))

(defun minroot (a b c)
  (if (zerop a)
      (/ (- c) b)
      (let ((disc (- (sq b) (* 4 a c))))
        (unless (minusp disc)
          (let ((discrt (sqrt disc)))
            (min (/ (+ (- b) discrt) (* 2 a))
                 (/ (- (- b) discrt) (* 2 a))))))))
```

Рис. 9.2. *Математические утилиты*

Наконец, функция minroot принимает три действительных числа a, b и c и возвращает наименьшее x, для которого $ax^2+bx+c=0$. В случае когда a не равно нулю, корни этого уравнения могут быть получены по хорошо известной формуле:

$$x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$$

Код, реализующий функционально ограниченный трассировщик лучей, представлен на рис. 9.3. Он генерирует черно-белые изображения, освещаемые одним источником света, расположенным там же, где и глаз наблюдателя. (Таким образом достигается эффект фотографии со вспышкой.)

```
(defstruct surface color)

(defparameter *world* nil)
(defconstant eye (make-point :x 0 :y 0 :z 200))

(defun tracer (pathname &optional (res 1))
  (with-open-file (p pathname :direction :output)
    (format p "P2 ~A ~A 255" (* res 100) (* res 100))
    (let ((inc (/ res)))
      (do ((y -50 (+ y inc)))
          ((< (- 50 y) inc))
        (do ((x -50 (+ x inc)))
            ((< (- 50 x) inc))
          (print (color-at x y) p))))))

(defun color-at (x y)
  (multiple-value-bind (xr yr zr)
                       (unit-vector (- x (x eye))
                                    (- y (y eye))
                                    (- 0 (z eye)))
    (round (* (sendray eye xr yr zr) 255))))

(defun sendray (pt xr yr zr)
  (multiple-value-bind (s int) (first-hit pt xr yr zr)
    (if s
        (* (lambert s int xr yr zr) (surface-color s))
        0)))

(defun first-hit (pt xr yr zr)
  (let (surface hit dist)
    (dolist (s *world*)
      (let ((h (intersect s pt xr yr zr)))
        (when h
          (let ((d (distance h pt)))
            (when (or (null dist) (< d dist))
              (setf surface s hit h dist d))))))
    (values surface hit)))

(defun lambert (s int xr yr zr)
  (multiple-value-bind (xn yn zn) (normal s int)
    (max 0 (+ (* xr xn) (* yr yn) (* zr zn)))))
```

Рис. 9.3. *Трассировка лучей*

Для представления объектов в виртуальном мире используется структура `surface`. Говоря точнее, она будет включена в структуры, представляющие конкретные разновидности объектов, например сферы. Сама структура `surface` содержит лишь одно поле `color` (цвет) в интервале от `0` (черный) до `1` (белый).

Плоскость рисунка располагается вдоль осей x и y. Глаз наблюдателя смотрит вдоль оси z и находится на расстоянии 200 единиц от рисунка.

Чтобы добавить объект, необходимо поместить его в список `*world*` (изначально `nil`). Чтобы он был видимым, его z-координата должна быть отрицательной. Рисунок 9.4 демонстрирует прохождение лучей через поверхность рисунка и их падение на сферу.

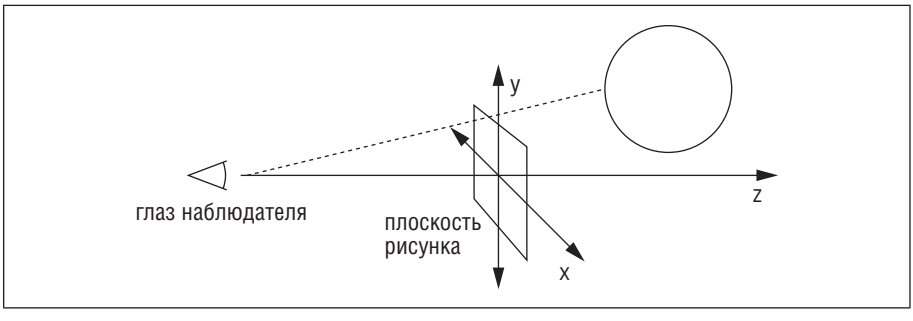

Рис. 9.4. *Трассировка лучей*

Функция `tracer` записывает изображение в файл по заданному пути. Запись производится в обычном ASCII-формате, называемом PGM. По умолчанию размер изображения – 100×100. Заголовок PGM-файла начинается с тега `P2`, за которым следуют числа, соответствующие ширине (100) и высоте (100) в пикселах, причем максимально возможное значение равно 255. Оставшаяся часть файла содержит 10000 целых чисел от 0 (черный) до 255 (белый), которые вместе дают 100 горизонтальных полос по 100 пикселов в каждой.

Разрешение изображения может быть изменено с помощью `res`. Если `res` равно, например, `2`, то изображение будет содержать 200×200 пикселов.

По умолчанию изображение – это квадрат 100×100 на плоскости рисунка. Каждый пиксел представляет собой количество света, проходящего через данную точку к глазу наблюдателя. Для нахождения этой величины `tracer` вызывает `color-at`. Эта функция ищет вектор от глаза наблюдателя к заданной точке, затем вызывает `sendray`, направляя луч вдоль этого вектора через моделируемый мир. В результате `sendray` получает некоторое число от 0 до 1, которое затем масштабируется на отрезок от 0 до 255.

Для определения интенсивности света `sendray` ищет объект, от которого он был отражен. Для этого вызывается функция `first-hit`, которая находит среди всех объектов в `*world*` тот, на который луч падает первым, или же убеждается в том, что луч не падает ни на один объект. В последнем случае возвращается цвет фона, которым мы условились считать `0` (черный). Если луч падает на один из объектов, то нам необходимо найти долю света, отраженного от него. Согласно закону Ламберта, интенсивность света, отраженного точкой поверхности, пропорциональна скалярному произведению единичного вектора N, направленного из этой

точки вдоль нормали к поверхности (вектор, длина которого равна 1, направленный перпендикулярно поверхности в заданной точке), и единичного вектора L, направленного вдоль луча к источнику света:

$$i = N \cdot L$$

Если источник света находится в этой точке, N и L будут иметь одинаковое направление, и их скалярное произведение будет иметь максимальное значение, равное 1. Если поверхность находится под углом 90° к источнику света, то N и L будут перпендикулярны и их произведение будет равно 0. Если источник света находится за поверхностью, то произведение будет отрицательным.

В нашей программе мы предполагаем, что источник света находится там же, где и глаз наблюдателя, поэтому функция `lambert`, использующая вышеописанное правило для нахождения освещенности некоторой точки поверхности, возвращает скалярное произведение нормали и трассируемого луча.

В функции `sendray` это число умножается на цвет поверхности (темная поверхность отражает меньше света), чтобы определить интенсивность в заданной точке. Упростим задачу, ограничившись лишь объектами типа сферы. Код, реализующий поддержку сфер, приведен на рис. 9.5. Структура `sphere` включает `surface`, поэтому `sphere` будет иметь параметр `color`, так же как и параметры `center` и `radius`. Вызов `defsphere` добавляет в наш мир новую сферу.

Функция `intersect` (пересечение) распознает тип объекта и вызывает соответствующую ему функцию. На данный момент мы умеем работать лишь со сферами. Для сферы из `intersect` будет вызываться `sphere-intersect`, но наш код может быть с легкостью расширен для поддержки других объектов.

Как найти пересечение луча со сферой? Луч представляется точкой $p=(x_0, y_0, z_0)$ и единичным вектором $v=(x_r, y_r, z_r)$. Любая точка луча может быть выражена как $p + nv$ для некоторого n или, что то же самое, $(x_0+nx_r, y_0+ny_r, z_0+nz_r)$. Когда луч падает на сферу, расстояние до центра (x_c, y_c, z_c) равно радиусу сферы r. Таким образом, условие пересечения луча и сферы можно записать следующим образом:

$$r = \sqrt{(x_0 + nx_r - x_c)^2 + (y_0 + ny_r - y_c)^2 + (z_0 + nz_r - z_c)^2}$$

Из этого следует, что

$$an^2 + bn + c = 0$$

где

$$a = x_r^2 + y_r^2 + z_r^2$$
$$b = 2\left((x_0 - x_c)x_r + (y_0 - y_c)y_r + (z_0 - z_c)z_r\right)$$
$$c = (x_0 - x_c)^2 + (y_0 - y_c)^2 + (z_0 - z_c)^2 - r^2$$

```
(defstruct (sphere (:include surface))
  radius center)

(defun defsphere (x y z r c)
  (let ((s (make-sphere
             :radius r
             :center (make-point :x x :y y :z z)
             :color  c)))
    (push s *world*)
    s))

(defun intersect (s pt xr yr zr)
  (funcall (typecase s (sphere #'sphere-intersect))
           s pt xr yr zr))

(defun sphere-intersect (s pt xr yr zr)
  (let* ((c (sphere-center s))
         (n (minroot (+ (sq xr) (sq yr) (sq zr))
                     (* 2 (+ (* (- (x pt) (x c)) xr)
                             (* (- (y pt) (y c)) yr)
                             (* (- (z pt) (z c)) zr)))
                     (+ (sq (- (x pt) (x c)))
                        (sq (- (y pt) (y c)))
                        (sq (- (z pt) (z c)))
                        (- (sq (sphere-radius s)))))))
    (if n
        (make-point :x  (+ (x pt) (* n xr))
                    :y  (+ (y pt) (* n yr))
                    :z  (+ (z pt) (* n zr))))))

(defun normal (s pt)
  (funcall (typecase s (sphere #'sphere-normal))
           s pt))

(defun sphere-normal (s pt)
  (let ((c (sphere-center s)))
    (unit-vector (- (x c) (x pt))
                 (- (y c) (y pt))
                 (- (z c) (z pt)))))
```

Рис. 9.5. *Сферы*

Для нахождения точки пересечения нам необходимо решить это квадратное уравнение. Оно может иметь ноль, один или два вещественных корня. Отсутствие решений означает, что луч проходит мимо сферы; одно решение означает, что луч пересекает сферу лишь в одной точке (то есть касается ее); два решения сигнализируют о том, что луч проходит через сферу, пересекая ее поверхность два раза. В последнем случае нам интересен лишь наименьший из корней. При удалении луча от глаза наблюдателя n увеличивается, поэтому наименьшее решение будет

соответствовать меньшему n. Для нахождения корня используется `min-root`. Если корень существует, `sphere-intersect` возвратит соответствующую ему точку $(x_0 + nx_r, y_0 + ny_r, z_0 + nz_r)$.

Две другие функции на рис. 9.5, `normal` и `sphere-normal`, связаны между собой аналогично `intersect` и `sphere-intersect`. Поиск нормали для сферы крайне прост – это всего лишь вектор, направленный из точки на поверхности к ее центру.

На рис. 9.6 показано, как будет выполняться генерация изображения; `ray-trace` создает 38 сфер (не все они будут видимыми), а затем генерирует изображение, которое записывается в файл `"spheres.pgm"`. Итог работы программы с параметром `res = 10` представлен на рис. 9.7.

```
(defun ray-test (&optional (res 1))
  (setf *world* nil)
  (defsphere 0 -300 -1200 200 .8)
  (defsphere -80 -150 -1200 200 .7)
  (defsphere 70 -100 -1200 200 .9)
  (do ((x -2 (1+ x)))
      ((> x 2))
    (do ((z 2 (1+ z)))
        ((> z 7))
      (defsphere (* x 200) 300 (* z -400) 40 .75)))
  (tracer (make-pathname :name "spheres.pgm") res))
```

Рис. 9.6. *Использование трассировщика*

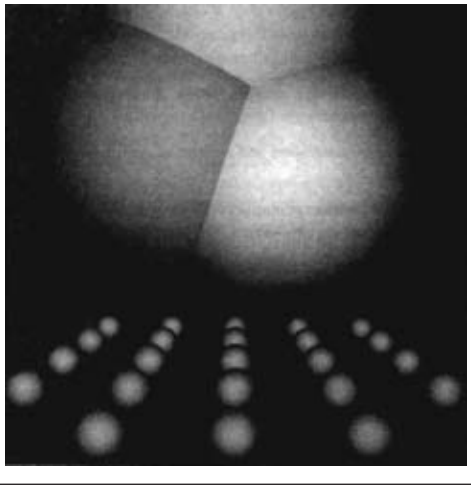

Рис. 9.7. *Изображение, полученное методом трассировки лучей*

Полноценный трассировщик лучей в состоянии генерировать куда более сложные изображения с учетом нескольких источников света разной интенсивности, расположенных в произвольных точках пространства. Для этого программа должна учитывать эффекты отбрасывания теней. Расположение источника света рядом с глазом наблюдателя позволило нам существенно упростить задачу, так как в этом случае ни одна из отбрасываемых теней не попадает в поле видимости.

Кроме того, полноценный трассировщик должен учитывать вторичные отражения. Разумеется, объекты различных цветов будут отражать свет по-разному. Тем не менее основной алгоритм, представленный на рис. 9.3, предоставляет механизм трассировки, который в дальнейшем может быть улучшен и дополнен, например с помощью рекурсивного использования уже имеющихся методов.

Кроме того, реальная программа-трассировщик должна быть тщательно оптимизирована. Наша программа имеет сжатые размеры и не оптимизирована ни как Лисп-программа, ни как трассировщик лучей. Добавление к программе одних только деклараций inline и деклараций типов (см. раздел 13.3) может обеспечить более чем двукратный прирост производительности.

Итоги главы

1. Common Lisp предоставляет в распоряжение целые числа, рациональные дроби, числа с плавающей запятой и комплексные числа.
2. Числовые значения могут быть преобразованы или упрощены, а их компоненты могут быть извлечены.
3. Предикаты сравнения чисел принимают любое количество аргументов и последовательно сравнивают их пары, за исключением `/=`, который сравнивает все возможные пары аргументов.
4. В Common Lisp представлены практически все функции, имеющиеся в низкоуровневом инженерном калькуляторе. Одни и те же функции могут применяться к аргументам любых типов.
5. Числа типа `fixnum` – небольшие целые числа, умещающиеся в одном машинном слове. При необходимости они без предупреждения преобразуются к типу `bignum`. Однако затраты на его использование существенно выше. Common Lisp также предоставляет до 4-х типов чисел с плавающей запятой. Для каждой реализации ограничения на их размер свои и могут быть получены из специальных констант.
6. Трассировщик лучей генерирует изображение, отслеживая прохождение света через смоделированный мир и вычисляя интенсивность излучения для каждого пиксела изображения.

Упражнения

1. Определите функцию, принимающую список действительных чисел и возвращающую истину, когда числа следуют в порядке неубывания.
2. Определите функцию, принимающую целочисленную величину (количество центов) и возвращающую четыре значения, показывающие, как собрать заданную сумму с помощью монет стоимостью 25, 10, 5 и 1 цент, используя наименьшее их количество.
3. На очень далекой планете живут два вида существ: вигглы и вобблы. И первые, и вторые одинаково хорошо поют. Каждый год на этой планете проводится грандиозное соревнование, по результатам которого выбирают десять лучших певцов. Вот результаты за прошедшие десять лет:

Год	1	2	3	4	5	6	7	8	9	10
Вигглы	6	5	6	4	5	5	4	5	6	5
Воблы	4	5	4	6	5	5	6	5	4	5

   Напишите программу, симулирующую подобные соревнования. Определите по результату вашей программы, действительно ли жюри каждый год выбирает десять самых лучших певцов.
4. Определите функцию, принимающую 8 действительных чисел, представляющих два отрезка в двумерном пространстве. Функция возвращает `nil`, если отрезки не пересекаются, в противном случае возвращаются два значения: x- и y-координаты точки их пересечения.

Пусть функция `f` имеет один действительный аргумент, а `min` и `max` – ненулевые действительные числа разных знаков; `f` имеет корень (возвращает ноль) для некоторого числа i, такого что `min` < i < `max`. Определите функцию, принимающую четыре аргумента `f`, `min`, `max` и `epsilon` и возвращающую приближенное значение i с точностью до `epsilon`.

5. *Метод Горнера* позволяет эффективно решать полиномы. Чтобы найти ax^3+bx^2+cx+d, нужно вычислить $x(x(ax+b)+c)+d$. Определите функцию, принимающую один или несколько аргументов – значение x и следующие за ним n действительных чисел, представляющих полином степени $(n-1)$. Функция должна вычислять значение полинома для заданного x методом Горнера.
6. Сколько бит использовала бы ваша реализация для представления `fixnum`?
7. Сколько различных типов чисел с плавающей запятой представлено в вашей реализации?

10

Макросы

Лисп-код записывается в виде списков, которые сами являются Лисп-объектами. В разделе 2.3 говорилось о том, что эта особенность предоставляет возможность писать программы, которые пишут другие программы. В этой главе показано, как переступить черту между выражениями и кодом.

10.1. Eval

Очевидно, что создавать выражения нужно путем вызова `list`. Однако возникает другой вопрос: как объяснить Лиспу, что созданный список – это код? Для этого существует функция `eval`, вычисляющая заданное выражение и возвращающая его значение:

```
> (eval '(+ 1 2 3))
6
> (eval '(format t "Hello"))
Hello
NIL
```

Да, именно об `eval` мы говорили все это время. Следующая функция реализует некое подобие toplevel:

```
(defun our-toplevel ()
  (do ()
      (nil)
    (format t "~%> ")
    (format (eval (read)))))
```

По этой причине toplevel также называют read-eval-print loop (цикл чтение-вычисление-печать).

Вызов `eval` – один из способов перешагнуть границу между списками и кодом. Однако это не лучший путь:

1. Он неэффективен: eval получает простой список, и она вынуждена его либо скомпилировать, либо вычислить в интерпретаторе. Несомненно, оба этих варианта намного медленнее, чем исполнение уже скомпилированного кода.
2. Выражение вычисляется вне какого-либо лексического контекста. Если, например, вызвать eval внутри let, выражения, переданные в eval, не смогут ссылаться на переменные, определенные в let.

Существуют гораздо более оптимальные способы генерации кода (см. описание в следующем разделе). А что касается eval, то ее можно использовать, пожалуй, лишь для создания чего-то вроде циклов toplevel.

Поэтому для программистов основная ценность eval заключается в том, что эта функция выражает концептуальную суть Лиспа. Можно считать, что eval определена как большое выражение cond:

```
(defun eval (expr env)
  (cond ...
        ((eql (car expr) 'quote) (cadr expr))
        ...
        (t (apply (symbol-function (car expr))
                  (mapcar #'(lambda (x)
                              (eval x env))
                          (cdr expr))))))
```

Большинство выражений обрабатываются последним условием, которое говорит: взять функцию, имя которой записано в car, вычислить аргументы из cdr и вернуть результат применения функции к хвосту (cdr) списка[1].

Тем не менее для выражения вида (quote x) такой подход неприменим, поскольку вся суть quote в том, чтобы защитить свой аргумент от вычисления. Это значит, что нам необходимо специальное условие для quote. Так мы приходим к сути понятия «специальный оператор» – это оператор, для которого в eval имеется особое условие.

Функции coerce и compile также предоставляют возможность перейти от списков к коду. Лямбда-выражение можно превратить в функцию:

```
> (coerce '(lambda (x) x) 'function)
#<Interpreted-Function BF9D96>
```

С помощью функции compile с первым аргументом nil также можно скомпилировать лямбда-выражение, переданное вторым аргументом:

```
> (compile nil '(lambda (x) (+ x 2)))
#<Compiled-Function DF55BE>
```

[1] Как и реальная eval, наша функция имеет еще один аргумент (env), представляющий лексическое окружение. Неточность нашей модели eval состоит в том, что она получает функцию перед тем, как приступает к обработке аргументов, в то время как в Common Lisp порядок выполнения этих операций специальным образом не определен.

```
NIL
NIL
```

Поскольку coerce и compile могут принимать списки в качестве аргументов, программа может создавать новые функции на лету. Однако этот метод также нельзя назвать лучшим, так как он имеет те же недостатки, что и eval.

Причина неэффективности eval, coerce и compile в том, что они пересекают границу между списками и кодом и создают функции в *момент выполнения* (*run-time*). Пересечение границы – дорогое удовольствие. В большинстве случаев это лучше делать во время компиляции, а не в процессе выполнения программы. В следующем разделе будет показано, как это сделать.

10.2. Макросы

Наиболее распространенный способ писать программы, которые будут писать программы, – это создание макросов. *Макросы* – это операторы, которые реализуются через трансформацию. Определяя макрос, вы указываете, как его вызов должен быть преобразован. Само преобразование автоматически выполняется компилятором и называется *раскрытием макроса* (*macro-expansion*). Код, полученный в результате раскрытия макроса, естественным образом становится частью программы, как если бы он был набран вами.

Обычно макросы создаются с помощью defmacro. Вызов defmacro напоминает вызов defun, но вместо того чтобы описать, как должно быть получено значение, он указывает, как этот вызов должен быть преобразован. Например, макрос, устанавливающий значение своего аргумента в nil, может быть определен так:

```
(defmacro nil! (x)
  (list 'setf x nil))
```

Данный макрос создает новый оператор nil!, который принимает один аргумент. Вызов вида (nil! *a*) будет преобразован в (setf *a* nil) и лишь затем скомпилирован или вычислен. Таким образом, если набрать (nil! x) в toplevel,

```
> (nil! x)
NIL
> x
NIL
```

это в точности эквивалентно набору выражения (setf x nil).

Чтобы протестировать функцию, ее вызывают. Чтобы протестировать макрос, его раскрывают. Функция macroexpand-1 принимает вызов макроса и возвращает результат его раскрытия:

```
> (macroexpand-1 '(nil! x))
(SETF X NIL)
T
```

Макрос может раскрываться в другой макровызов. Когда компилятор (или toplevel) встречает такой макровызов, он раскрывает этот макрос до тех пор, пока в коде не останется ни одного другого макровызова.

Секрет понимания макросов в осознании их реализации. Они – не более чем функции, преобразующие выражения. К примеру, если вы передадите выражение вида (nil! *a*) в следующую функцию:

```
(lambda (expr)
  (apply #'(lambda (x) (list 'setf x nil))
         (cdr expr)))
```

то она вернет (setf *a* nil). Применяя defmacro, вы выполняете похожую работу. Все, что делает macroexpand-1, когда встречает выражение, car которого соответствует имени одного из макросов, – это перенаправляет это выражение в соответствующую функцию.

10.3. Обратная кавычка

Макрос чтения *обратная кавычка* (*backquote*) позволяет строить списки на основе специальных шаблонов. Она широко используется при определении макросов. Обычной кавычке на клавиатуре соответствует закрывающая прямая кавычка (апостроф), а обратной – открывающая. Ее называют обратной из-за ее наклона влево.

Обратная кавычка, использованная сама по себе, эквивалентна обычной кавычке:

```
> `(a b c)
(A B C)
```

Как и обычная кавычка, обратная кавычка предотвращает вычисление выражения.

Преимуществом обратной кавычки является возможность выборочного вычисления частей выражения с помощью , (запятой) и ,@ (запятая-эт). Любое подвыражение, предваряемое запятой, будет вычислено. Сочетая обратную кавычку и запятую, можно строить шаблоны списков:

```
> (setf a 1 b 2)
2
> `(a is ,a and b is ,b)
(A IS 1 AND B IS 2)
```

Используя обратную кавычку вместо вызова list, мы можем записывать определения макросов, которые будут выглядеть так же, как результат их раскрытия. К примеру, оператор nil! может быть определен следующим образом:

```
(defmacro nil! (x)
  `(setf ,x nil))
```

Запятая-эт действует похожим образом, но вставляет поэлементно свой аргумент (который должен быть списком) в середину другого списка:

```
> (setf lst '(a b c))
(A B C)
> `(lst is ,lst)
(LST IS (A B C))
> `(its elements are ,@lst)
(ITS ELEMENTS ARE A B C)
```

Это может оказаться полезным для макросов, которые используют остаточный аргумент, например, для представления тела кода. Предположим, мы хотим создать макрос `while`, который вычисляет свои аргументы до тех пор, пока проверочное выражение остается истинным:

```
> (let ((x 0))
    (while (< x 10)
      (princ x)
      (incf x)))
0123456789
NIL
```

Мы можем определить такой макрос, используя остаточный параметр, который соберет все выражения тела макроса в список, а затем этот список будет встроен в результат раскрытия макроса благодаря `,@`:

```
(defmacro while (test &rest body)
  `(do ()
       ((not ,test))
     ,@body))
```

10.4. Пример: быстрая сортировка

На рис. 10.1 приведен пример[1] функции, полностью построенной на макросах, – функции, выполняющей сортировку векторов с помощью алгоритма быстрой сортировки (quicksort).°

Алгоритм быстрой сортировки выполняется следующим образом:

1. Один из элементов принимается в качестве *опорного*. Многие реализации выбирают элемент из середины вектора.
2. Затем производится *разбиение* вектора, при этом его элементы переставляются местами до тех пор, пока все элементы, меньшие опорного, не будут находиться по левую сторону от больших или равных ему.
3. Наконец, если хотя бы одна из частей состоит из двух и более элементов, алгоритм применяется рекурсивно к каждому из сегментов.

[1] Код на рис. 10.1 содержит слегка исправленный код. Ошибка найдена Джедом Кросби. – *Прим. перев.*

```
(defun quicksort (vec l r)
  (let ((i l)
        (j r)
        (p (svref vec (round (+ l r) 2))))    ; 1
    (while (<= i j)                           ; 2
      (while (< (svref vec i) p) (incf i))
      (while (> (svref vec j) p) (decf j))
      (when (<= i j)
        (rotatef (svref vec i) (svref vec j))
        (incf i)
        (decf j)))
    (if (>= (- j l) 1) (quicksort vec l j))   ; 3
    (if (>= (- r i) 1) (quicksort vec i r)))
  vec)
```

Рис. 10.1. *Быстрая сортировка*

С каждым последующим вызовом сортируемый список становится короче, и так до тех пор, пока весь список не будет отсортирован.

Реализация этого алгоритма (рис. 10.1) требует указания вектора и двух чисел, задающих диапазон сортировки. Элемент, находящийся в середине этого диапазона, выбирается в качестве опорного элемента (`p`). Затем по мере продвижения вдоль вектора к окончанию диапазона производятся перестановки его элементов, которые либо слишком большие, чтобы находиться слева, либо слишком малые, чтобы находиться справа. (Перестановка двух элементов осуществляется с помощью `rotatef`.) Далее эта процедура повторяется рекурсивно для каждой полученной части, содержащей более одного элемента.

Помимо макроса `while`, определенного в предыдущем разделе, в `quicksort` (рис. 10.1) задействованы встроенные макросы `when`, `incf`, `decf` и `rotatef`. Их использование существенно упрощает код, делая его аккуратным и понятным.

10.5. Проектирование макросов

Написание макросов – это отдельная разновидность программирования, имеющая собственные уникальные цели и проблемы. Умение изменять поведение компилятора отчасти похоже на возможность изменять сам компилятор. Поэтому при разработке макросов нужно мыслить как создатель языка.

В этом разделе будут рассмотрены основные трудности, связанные с написанием макросов, и методики их разрешения. В качестве примера мы создадим макрос `ntimes`, вычисляющий свое тело *n* раз:

```
> (ntimes 10
    (princ "."))
```

```
..........
NIL
```

Ниже приведено некорректное определение макроса ntimes, иллюстрирующее некоторые проблемы, связанные с проектированием макросов:

```
(defmacro ntimes (n &rest body)            ; wrong
  `(do ((x 0 (+ x 1)))
       ((>= x ,n))
     ,@body))
```

На первый взгляд такое определение кажется вполне корректным. Для приведенного выше примера оно будет работать нормально, однако на самом деле в нем есть две проблемы.

Первой проблемой, о которой должны думать разработчики макросов, является непреднамеренный *захват переменных*. Это случается, когда имя переменной, используемой в раскрытии макроса, совпадает с именем переменной, которая уже существует в том окружении, куда вставляется раскрытие макроса. Некорректное определение ntimes создает переменную с именем x. Поэтому если макрос вызывается там, где уже существует переменная с таким же именем, результат его выполнения может не соответствовать нашим ожиданиям:

```
> (let ((x 10))
    (ntimes 5
      (setf x (+ x 1)))
    x)
10
```

Мы предполагали, что значение x будет увеличено в пять раз и в итоге будет равно 15. Но поскольку переменная x используется еще и внутри макроса как итерируемая переменная в do, setf будет увеличивать значение *этой* переменной, а не той, что предполагалось. Раскрыв этот макрос, мы увидим, что предыдущее выражение выглядит следующим образом:

```
(let ((x 10))
  (do ((x 0 (+ x 1)))
      ((> x 5))
    (setf x (+ x 1)))
  x)
```

Наиболее общим решением будет не использовать обычные символы в тех местах, где они могут быть захвачены. Вместо этого можно использовать gensym (см. раздел 8.4). В связи с тем что read интернирует каждый встреченный символ, gensym никоим образом не будет равен (с точки зрения eql) любому другому символу, встречающемуся в тексте программы. Переписав наше определение ntimes с использованием gensym вместо x, мы сможем избавиться от вышеописанной проблемы:

```
(defmacro ntimes (n &rest body)            ; wrong
  (let ((g (gensym)))
```

```
    `(do ((,g 0 (+ ,g 1)))
         ((>= ,g ,n))
       ,@body)))
```

Тем не менее в данном определении по-прежнему кроется другая проблема: повторное вычисление. Поскольку первый аргумент встраивается непосредственно в вызов do, он будет вычисляться на каждой итерации. Эта ошибка ясно демонстрируется примером, в котором первое выражение содержит побочные эффекты:

```
> (let ((v 10))
    (ntimes (setf v (- v 1))
      (princ ".")))
.....
NIL
```

Поскольку начальное значение v равно 10 и setf возвращает значение своего второго аргумента, мы ожидали получить девять точек, однако в действительности их только пять.

Чтобы разобраться с причиной такого эффекта, посмотрим на результат раскрытия макроса:

```
(let ((v 10))
  (do ((#:g1 0 (+ #:g1 1)))
      ((>= #:g1 (setf v (- v 1))))
    (princ ".")))
```

На каждой итерации мы сравниваем значение переменной (gensym при печати обычно предваряется #:) не с числом 9, а с выражением, которое уменьшается на единицу при каждом вызове.

Нежелательных повторных вычислений можно избежать, вычислив значение выражения перед началом цикла и записав его в переменную. Обычно для этого требуется еще один gensym:

```
(defmacro ntimes (n &rest body)
  (let ((g (gensym))
        (h (gensym)))
    `(let ((,h ,n))
       (do ((,g 0 (+ ,g 1)))
           ((>= ,g ,h))
         ,@body))))
```

Эта версия ntimes является полностью корректной.

Непреднамеренные повторные вычисления и захват переменных – основные проблемы, возникающие при разработке макросов, но есть и другие. С опытом видеть ошибки в макросах станет так же легко и естественно, как предупреждать деление на ноль. Но поскольку макросы наделяют нас новыми возможностями, то и сложности, с которыми нам придется столкнуться, тоже новы.

Ваша реализация Common Lisp содержит множество примеров, на которых вы можете поучиться проектировать макросы. Раскрывая вызовы

встроенных макросов, вы сможете понять их устройство. Ниже приведен результат раскрытия cond для большинства реализаций:

```
> (pprint (macroexpand-1 '(cond (a b)
                                (c d e)
                                (t f))))
(IF A
    B
    (IF C
        (PROGN D E)
        F))
```

Функция pprint, печатающая выражения с необходимыми отступами, сделает выводимый код легкочитаемым.

10.6. Обобщенные ссылки

Поскольку макровызов раскрывается непосредственно в том месте кода, где он был вызван, любой макровызов, раскрывающийся в выражение, которое может быть первым аргументом setf, сам может быть первым аргументом setf. Определим, например, синоним car:

```
(defmacro cah (lst) `(car ,lst))
```

Так как выражение с car может быть первым аргументом setf, то и аналогичный вызов с cah также может иметь место:

```
> (let ((x (list 'a 'b 'c)))
    (setf (cah x) 44)
    x)
(44 B C)
```

Написание макроса, который самостоятельно раскрывается в вызов setf, – менее тривиальная задача, несмотря на кажущуюся простоту. На первый взгляд можно определить incf, например так:

```
(defmacro incf (x &optional (y 1))         ; wrong
  `(setf ,x (+ ,x ,y)))
```

Но такое определение не работает. Следующие два выражения не эквивалентны:

```
(setf (car (push 1 lst)) (1+ (car (push 1 lst))))

(incf (car (push 1 lst)))
```

Если lst изначально равно nil, то второе выражение установит его в (2), тогда как первое выражение установит его в (1 2).

Common Lisp предоставляет define-modify-macro как способ написания ограниченного класса макросов поверх setf. Он принимает три аргумента: имя макроса, дополнительные аргументы (место, подлежащее изменению, является неявным первым аргументом) и имя функции, которая

вернет новое значение заданного места. Теперь мы можем определить incf так:

```
(define-modify-macro our-incf (&optional (y 1)) +)
```

А вот версия push для добавления элемента в конец списка:

```
(define-modify-macro append1f (val)
  (lambda (lst val) (append lst (list val))))
```

Последний макрос работает следующим образом:

```
> (let ((lst '(a b c)))
    (append1f lst 'd)
    lst)
(A B C D)
```

Между прочим, макросы push и pop не могут быть определены через define-modify-macro, так как в push изменяемое место не является первым аргументом, а в случае pop его возвращаемое значение не является измененным объектом.

10.7. Пример: макросы-утилиты

В разделе 6.4 рассматривалась концепция утилит, операторов общего назначения. С помощью макросов мы можем создавать утилиты, которые не могут быть созданы как функции. Несколько подобных примеров мы уже видели: nil!, ntimes, while. Все они позволяют управлять процессом вычисления аргументов, а поэтому могут быть реализованы только в виде макросов. В этом разделе вы найдете больше примеров утилит-макросов. На рис. 10.2 приведена выборка макросов, которые на практике доказали свое право на существование.

Первый из них, for, по устройству напоминает while (стр. 174). Его тело вычисляется в цикле, каждый раз для нового значения переменной:

```
> (for x 1 8
    (princ x))
12345678
NIL
```

Выглядит несомненно проще, чем аналогичное выражение с do:

```
(do ((x 1 (1+ x)))
    ((> x 8))
  (princ x))
```

Результат раскрытия выражения с for будет очень похож на выражение с do:

```
(do ((x 1 (1+ x))
     (#:g1 8))
    ((> x #:g1))
  (princ x))
```

```
(defmacro for (var start stop &body body)
  (let ((gstop (gensym)))
    `(do ((,var ,start (1+ ,var))
          (,gstop ,stop))
         ((> ,var ,gstop))
       ,@body)))

(defmacro in (obj &rest choices)
  (let ((insym (gensym)))
    `(let ((,insym ,obj))
       (or ,@(mapcar #'(lambda (c) `(eql ,insym ,c))
                     choices)))))

(defmacro random-choice (&rest exprs)
  `(case (random ,(length exprs))
     ,@(let ((key -1))
         (mapcar #'(lambda (expr)
                     `(,(incf key) ,expr))
                 exprs))))

(defmacro avg (&rest args)
  `(/ (+ ,@args) ,(length args)))

(defmacro with-gensyms (syms &body body)
  `(let ,(mapcar #'(lambda (s)
                     `(,s (gensym)))
                 syms)
     ,@body))

(defmacro aif (test then &optional else)
  `(let ((it ,test))
     (if it ,then ,else)))
```

Рис. 10.2. *Утилиты на макросах*

Макрос использует дополнительную переменную для хранения верхней границы цикла. В противном случае выражение `8` вычислялось бы каждый раз, а для более сложных случаев это нежелательно. Дополнительная переменная создается с помощью `gensym`, что позволяет предотвратить непреднамеренный захват переменной.

Второй макрос на рис. 10.2, `in`, возвращает истину, если его первый аргумент равен (с точки зрения `eql`) хотя бы одному из остальных своих аргументов. Без этого макроса вместо выражения:

```
(in (car expr) '+ '- '*)
```

нам пришлось бы писать:

```
(let ((op (car expr)))
  (or (eql op '+)
      (eql op '-)
      (eql op '*)))
```

Разумеется, первое выражение раскрывается в подобное этому, за исключением того, что вместо переменной op используется gensym.

Следующий пример, random-choice, случайным образом выбирает один из своих аргументов и вычисляет его. С задачей случайного выбора мы уже сталкивались (стр. 89). Макрос random-choice реализует его обобщенный механизм. Вызов типа:

```
(random-choice (turn-left) (turn-right))
```

раскрывается в:

```
(case (random 2)
  (0 (turn-left))
  (1 (turn-right)))
```

Следующий макрос, with-gensyms, предназначен в первую очередь для использования внутри других макросов. Нет ничего необычного в его применении в макросах, которые вызывают gensym для нескольких переменных. С его помощью вместо:

```
(let ((x (gensym)) (y (gensym)) (z (gensym)))
  ...)
```

мы можем написать:

```
(with-gensyms (x y z)
  ...)
```

Пока что ни один из приведенных выше макросов не мог быть написан в виде функции. Из этого следует правило: макрос должен создаваться лишь тогда, когда вы не можете написать аналогичную функцию. Однако из этого правила бывают исключения. Например, иногда вы можете определить оператор как макрос, чтобы иметь возможность выполнять какую-либо работу на этапе компиляции. Примером может служить макрос avg, вычисляющий среднее значение своих аргументов:

```
> (avg 2 4 8)
14/3
```

Аналогичная функция будет иметь вид:

```
(defun avg (&rest args)
  (/ (apply #'+ args) (length args)))
```

но эта функция вынуждена считать количество аргументов в процессе исполнения. Если нам вдруг захочется избежать этого, почему бы не вычислять длину списка аргументов (вызовом length) на этапе компиляции?

Последний макрос на рис. 10.2, aif, приводится в качестве примера преднамеренного захвата переменной. Он позволяет ссылаться через переменную it на значение, возвращаемое тестовым аргументом if. Благодаря этому вместо:

```
(let ((val (calculate-something)))
  (if val
      (1+ val)
      0))
```

мы можем написать:

```
(aif (calculate-something)
     (1+ it)
     0)
```

Разумное использование захвата переменных в макросах может оказать полезную услугу. Сам Common Lisp прибегает к такой возможности в некоторых случаях, например в `next-method-p` и `call-next-method`.

Разобранные в этом разделе макросы демонстрируют смысл фразы «программы, которые пишут программы». Единожды определенный `for` позволяет избежать написания множества однотипных выражений с `do`. Есть ли смысл в написании макроса просто для экономии на вводе? Однозначно. Ровно по этой же причине мы пользуемся языками программирования, заставляя компиляторы самостоятельно генерировать машинный код, вместо того чтобы набирать его вручную. Макросы позволяют вам дать своим программам те же преимущества, которые высокоуровневые языки дают программированию в целом. Аккуратное применение макросов позволит существенно сократить размер вашей программы, облегчая тем самым ее написание, понимание и поддержку.

Если у вас все еще остались какие-либо сомнения, представьте ситуацию, когда встроенные макросы недоступны и код, в который они раскрывались бы, необходимо набирать вручную. Теперь подойдите к вопросу с другой стороны. Представьте, что вы пишете программу, содержащую множество сходных частей. Задайте себе вопрос: «А не пишу ли я результаты раскрытия макросов?» Если так, то макросы, генерирующие этот код, – вот то, что вам нужно писать в действительности.

10.8. На Лиспе

Теперь, когда мы узнали, что такое макросы, мы видим, что даже большая часть самого Лиспа, чем мы предполагали, написана на самом Лиспе с помощью макросов. Большинство операторов в Common Lisp, которые не являются функциями, – это макросы, и все они написаны на Лиспе. И лишь 25 встроенных операторов являются специальными.

Джон Фодераро назвал Лисп «программируемым языком программирования».° С помощью функций и макросов Лисп может быть превращен в практически любой другой язык. (Подобный пример вы найдете в главе 17.) Каким бы ни был наилучший путь написания программы, можете быть уверены, что сможете подогнать Лисп под него.

Макросы являются одним из ключевых ингредиентов гибкости Лиспа. Они позволяют не просто изменять язык до неузнаваемости, но и делать

это понятным и эффективным способом. Интерес к макросам внутри Лисп-сообщества со временем только растет. Очевидно, что с их помощью уже сейчас можно делать удивительные вещи, но многие из них еще только предстоит обнаружить. Вам, например, если захотите. В руках программиста Лисп непрерывно эволюционирует. Именно поэтому он до сих пор жив.

Итоги главы

1. Вызов `eval` – один из способов использовать списки как код, но он неэффективен и не нужен.
2. Чтобы определить макрос, вам необходимо описать то, во что он будет раскрываться. На самом деле, макросы – это функции, возвращающие выражения.
3. Тело макроса, определенное с помощью обратной кавычки, выглядит как результат его раскрытия.
4. Разработчик макроса должен помнить о захвате переменных и повторных вычислениях. Чтобы протестировать макрос, полезно изучить результат его раскрытия, напечатанный с помощью `pprint`.
5. Проблема повторных вычислений возникает в большинстве макросов, раскрывающихся в вызовы `setf`.
6. Макросы более гибки по сравнению с функциями и позволяют создавать более широкий круг утилит. Вы даже можете извлекать пользу из возможности захвата переменных.
7. Лисп дожил до настоящих дней лишь потому, что эволюционирует в руках программиста. Это возможно в первую очередь благодаря макросам.

Упражнения

1. Пусть `x = a`, `y = b`, `z = (c d)`. Запишите выражения с обратной кавычкой, содержащие только заданные переменные (`x`, `y` и `z`) и приводящие к следующим результатам:

   (a) `((C D) A Z)`
   (b) `(X B C D)`
   (c) `((C D A) Z)`

2. Определите `if` через `cond`.
3. Определите макрос, аргументами которого являются число *n* и следующие за ним произвольные выражения. Макрос должен возвращать значение *n*-го выражения:

```
> (let ((n 2))
    (nth-expr n (/ 1 0) (+ 1 2) (/ 1 0)))
3
```

4. Определите `ntimes` (стр. 175), раскрывающийся в (локальную) рекурсивную функцию вместо вызова `do`.
5. Определите макрос `n-of`, принимающий на вход число *n* и выражение и возвращающий список значений, полученных последовательным вычислением этого выражения для возрастающего *n* раз значения переменной:

```
> (let ((i 0) (n 4))
    (n-of n (incf i)))
(1 2 3 4)
```

6. Определите макрос, принимающий список переменных и тело кода. По завершении выполнения кода переменные должны принять свои исходные значения.
7. Что не так со следующим определением `push`?

```
(defmacro push (obj lst)
  `(setf ,lst (cons ,obj ,lst)))
```

   Приведите пример ситуации, в которой данный код будет работать не так, как настоящий `push`.
8. Определите макрос, удваивающий свой аргумент:

```
> (let ((x 1))
    (double x)
    x)
2
```

11

CLOS

Объектная система Common Lisp (Common Lisp Object System, CLOS) представляет собой набор операторов для объектно-ориентированного программирования. Всех их объединяет историческое прошлое.° С технической точки зрения они ничем не отличаются от остального Common Lisp: `defmethod` – такая же неотъемлемая часть языка, как и `defun`.

11.1. Объектно-ориентированное программирование

Объектно-ориентированное программирование – это определенное изменение в организации программ. Это изменение подобно тому, которое произошло в распределении процессорного времени. В 1970 году под многопользовательской системой понимался один или два больших мейнфрейма, с которыми соединялось множество простых терминалов. Теперь же под этим принято понимать большой набор рабочих станций, объединенных в сеть. Таким образом, вычислительные мощности теперь распределены между индивидуальными участниками сети, а не централизованы в одном большом компьютере.

Объектно-ориентированное программирование ломает традиционное устройство программ похожим образом. Вместо реализации одной программы, управляющей всеми данными, самим этим данным объясняется, как вести себя, а собственно программа скрывается во взаимодействии этих новых «объектов» данных.

Допустим, мы хотим написать программу, вычисляющую площадь разнообразных двумерных фигур. Одним из подходов могло бы быть написание функции, которая выясняет тип объекта и ведет себя соответствующим образом. Пример такой функции приведен на рис. 11.1.

```
(defstruct rectangle
  height width)

(defstruct circle
  radius)

(defun area (x)
  (cond ((rectangle-p x)
         (* (rectangle-height x) (rectangle-width x)))
        ((circle-p x)
         (* pi (expt (circle-radius x) 2)))))

> (let ((r (make-rectangle)))
    (setf (rectangle-height r) 2
          (rectangle-width  r) 3)
    (area r))
6
```

Рис. 11.1. *Представление площадей через структуры и функции*

С помощью CLOS мы можем переписать код в другом ключе, как показано на рис. 11.2. В объектно-ориентированной модели программа разбивается на несколько *методов*, каждый из которых предназначен для соответствующего типа аргумента. Два метода, показанные на рис. 11.2, неявно определяют функцию `area`, которая будет вести себя так же, как аналогичная функция на рис. 11.1. При вызове `area` Лисп вызывает определенный метод в соответствии с типом аргумента.

Помимо разбиения функций на методы объектно-ориентированное программирование предполагает *наследование* – как для слотов, так и для методов. Пустой список, переданный вторым аргументом в двух вызовах `defclass` (рис. 11.2), – это список суперклассов (родительских классов). Определим теперь новый класс окрашенных объектов, а затем класс окрашенных кругов, имеющий два суперкласса: `circle` и `colored`:

```
(defclass colored ()
  (color))

(defclass colored-circle (circle colored)
  ())
```

Создавая экземпляры класса `colored-circle`, мы увидим два вида наследования:

1. Экземпляры `colored-circle` будут иметь два слота: `radius`, наследуемый из класса `circle`, и `color` – из класса `colored`.
2. Так как для класса `colored-circle` свой метод не определен, вызов `area` для данного класса будет использовать метод, определенный для класса `circle`.

```
(defclass rectangle ()
  (height width))

(defclass circle ()
  (radius))

(defmethod area ((x rectangle))
  (* (slot-value x 'height) (slot-value x 'width)))

(defmethod area ((x circle))
  (* pi (expt (slot-value x 'radius) 2)))

> (let ((r (make-instance 'rectangle)))
    (setf (slot-value r 'height) 2
          (slot-value r 'width)  3)
    (area r))
6
```

Рис. 11.2. *Представление площадей через классы и методы*

С практической точки зрения под объектно-ориентированным программированием подразумевается способ организации программы в терминах методов, классов, экземпляров и наследования. Зачем может понадобиться подобная организация? Одно из заявлений объектно-ориентированного подхода заключается в том, что он упрощает изменения в программах. Если мы хотим изменить способ отображения объектов класса `ob`, нам достаточно внести изменения лишь в один метод `display` этого класса. Если нам нужен новый класс объектов, похожих на `ob`, но слегка отличающихся от них, мы можем создать подкласс `ob`, для которого зададим необходимые нам свойства, а все остальное унаследуется от родительского класса. А если мы просто хотим получить один объект `ob`, который ведет себя отлично от остальных, мы можем создать нового потомка `ob` и напрямую поменять его свойства. Для внесения изменений в грамотно написанную программу нам необязательно даже смотреть на остальной код.°

11.2. Классы и экземпляры

В разделе 4.6 при создании структур мы проходили два этапа: с помощью `defstruct` создавали определение структуры, затем с помощью специальной функции `make-point` создавали саму структуру. Создание экземпляров требует двух аналогичных шагов. Для начала определяем *класс* с помощью `defclass`:

```
(defclass circle ()
  (radius center))
```

Только что мы определили класс `circle`, который имеет два *слота* (подобно полям структуры), названные `radius` и `center`.

Чтобы создать экземпляр этого класса, вместо вызова специфичной функции мы воспользуемся общей для всех классов функцией make-instance, вызванной с именем класса в качестве первого аргумента:

```
> (setf c (make-instance 'circle))
#<Circle #XC27496>
```

Доступ к значению слота можно осуществить с помощью slot-value. Новое значение можно установить с помощью setf:

```
> (setf (slot-value c 'radius) 1)
1
```

Как и для структур, значения неинициализированных слотов не определены.

11.3. Свойства слотов

Третий аргумент defclass должен содержать список определений слотов. Простейшим определением слота, как в нашем примере, служит символ, представляющий имя слота. В общем случае определение может быть списком, содержащим имя и набор свойств, передаваемых по ключу.

Параметр :accessor неявно создает функцию, обеспечивающую доступ к слоту, убирая необходимость использовать slot-value. Если мы обновим наше определение класса circle следующим образом:

```
(defclass circle ()
  ((radius :accessor circle-radius)
   (center :accessor circle-center)))
```

то сможем ссылаться на слоты с помощью функций circle-radius и circle-center соответственно:

```
> (setf c (make-instance 'circle))
#<Circle #XC5C726>
> (setf (circle-radius c) 1)
1
> (circle-radius c)
1
```

Если вместо :accessor использовать параметры :writer или :reader, можно задать по отдельности либо функцию установки значения слота, либо функцию чтения.

Исходное значение слота определяется параметром :initform. Чтобы можно было инициализировать слот в вызове make-instance по ключу, нужно задать имя соответствующего ключа как :initarg этого слота.[1] Определение класса, обладающего перечисленными свойствами, будет выглядеть следующим образом:

[1] В качестве имен используются, как правило, ключевые слова, хотя это не является обязательным требованием.

```
(defclass circle ()
  ((radius :accessor circle-radius
           :initarg :radius
           :initform 1)
   (center :accessor circle-center
           :initarg :center
           :initform (cons 0 0))))
```

Теперь вновь созданный экземпляр будет иметь установленные через :initform значения слотов, если с помощью :initarg не заданы другие значения:

```
> (setf c (make-instance 'circle :radius 3))
#<Circle #XC2DE0E>
> (circle-radius c)
3
> (circle-center c)
(0 . 0)
```

Обратите внимание, что :initarg имеет приоритет перед :initform.

Слоты могут быть *разделяемыми*, то есть имеющими одинаковое значение для всех экземпляров класса. Слот можно сделать разделяемым с помощью декларации :allocation :class. (Другой возможный вариант – :allocation :instance, но, поскольку это значение используется по умолчанию, задавать его нет смысла.) Если изменить значение такого слота в одном экземпляре, это приведет к изменению значения слота и во всех остальных экземплярах данного класса. Поэтому использование разделяемых слотов обосновано в том случае, когда класс имеет свойство, одинаковое для всех его экземпляров.

К примеру, предположим, что мы хотим смоделировать процесс наполнения содержимым желтой прессы. В нашей модели появление новой темы в одном таблоиде тут же приведет к ее появлению и в остальных, и этот факт нам нужно отразить в нашей программе. Этого можно добиться, если сделать соответствующий теме слот разделяемым. Определим класс tabloid следующим образом:

```
(defclass tabloid ()
  ((top-story :accessor tabloid-story
              :allocation :class)))
```

Если какая-либо тема попадает на первую страницу одного экземпляра таблоида, она тут же попадает на первую страницу другого:

```
> (setf daily-blab       (make-instance 'tabloid)
        unsolicited-mail (make-instance 'tabloid))
#<Tabloid #XC2AB16>
> (setf (tabloid-story daily-blab) 'adultery-of-senator)
ADULTERY-OF-SENATOR
> (tabloid-story unsolicited-mail)
ADULTERY-OF-SENATOR
```

Если задано необязательное свойство :documentation, то оно должно содержать строку описания данного слота. Задавая :type, вы обязуетесь обеспечить то, что слот будет содержать лишь значения заданного типа. Декларации типов будут рассмотрены в разделе 13.3.

11.4. Суперклассы

Второй аргумент defclass является списком *суперклассов*. От них класс наследует набор слотов. Так, если мы определим класс screen-circle как подкласс circle и graphic:

```
(defclass graphic ()
  ((color   :accessor graphic-color   :initarg :color)
   (visible :accessor graphic-visible :initarg :visible
            :initform t)))

(defclass screen-circle (circle graphic)
  ())
```

то экземпляры screen-circle будут иметь четыре слота, по два от каждого класса-родителя. И при этом не нужно создавать какие-либо новые слоты как его собственные – подкласс screen-circle создается лишь для объединения возможностей двух классов circle и graphic.

Параметры слотов (:accessor, :initarg и другие) наследуются вместе с самими слотами и работают так же, как если бы они использовались для экземпляров circle или graphic:

```
> (graphic-color (make-instance 'screen-circle
                                :color 'red :radius 3))
RED
```

Тем не менее в определении класса можно указывать некоторые параметры для наследуемых слотов, например :initform:

```
(defclass screen-circle (circle graphic)
  ((color :initform 'purple)))
```

Теперь экземпляры screen-circle будут пурпурными по умолчанию:

```
> (graphic-color (make-instance 'screen-color))
PURPLE
```

11.5. Предшествование

Мы познакомились с ситуацией, когда классы могут иметь несколько суперклассов. Если для некоторых суперклассов определен метод с одним и тем же именем, Лисп должен выбрать один из них, руководствуясь правилом *предшествования*.

Для каждого класса существует *список предшествования* – порядок следования классов от наиболее специфичного до наименее специфичного, включая рассматриваемый класс. В примерах, приведенных ранее, по-

рядок предшествования был тривиальным, однако он может усложниться в больших программах. Вот пример более сложной иерархии классов:

```
(defclass sculpture () (height width depth))

(defclass statue (sculpture) (subject))

(defclas metalwork () (metal-type))

(defclass casting (metalwork) ())

(defclass cast-statue (statue casting) ())
```

Граф, представляющий класс `cast-statue` и его суперклассы, изображен на рис. 11.3.

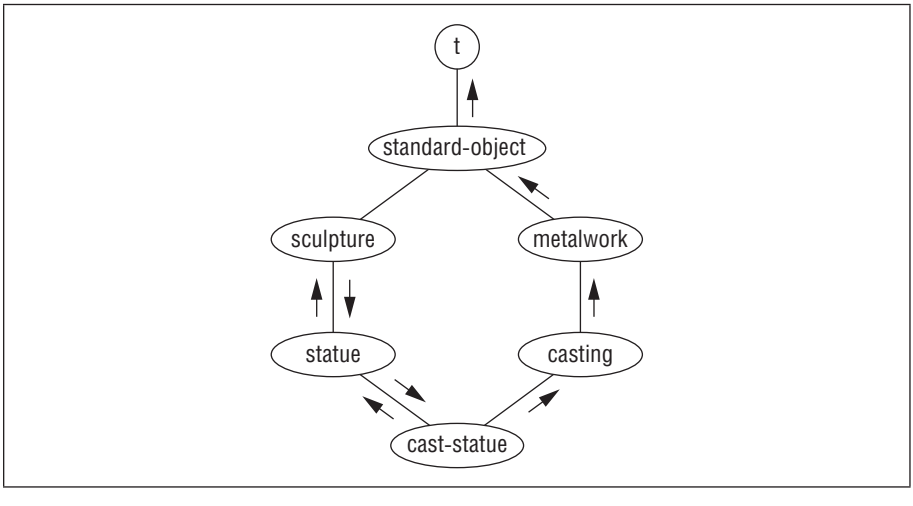

Рис. 11.3. *Иерархия классов*

Чтобы построить такой граф, необходимо начать с узла, соответствующего классу, и двигаться снизу-вверх. Ребра, направленные вверх, связывают класс с прямыми суперклассами, которые располагаются слева-направо в соответствии с порядком соответствующих вызовов `defclass`. Эта процедура повторяется до тех пор, пока для каждого класса будет существовать лишь один суперкласс – `standard-object`. Это справедливо для классов, в определении которых второй аргумент был `()`. Узел, соответствующий каждому такому классу, связан с узлом `standard-object`, а он в свою очередь – с классом `t`. Проделав подобные операции, мы получим граф, изображенный на рис. 11.3.

Теперь мы можем составить список предшествования:

1. Начинаем с низа графа.
2. Движемся вверх, всегда выбирая левое из ранее непройденных ребер.

3. Если обнаруживается, что мы пришли к узлу, к которому ведет ребро справа, то отматываем все перемещения до тех пор, пока не вернемся в узел с другими непройденными ребрами. Вернувшись, повторяем шаг 2.
4. Завершаем процедуру по достижении t. Порядок, в котором каждый узел был впервые посещен, определяет список предшествования.

В пункте 3 упоминается важный момент: ни один класс не может находиться в списке предшествования до любого из его подклассов.

Стрелки на рис. 11.3 показывают направление обхода. Список предшествования нашей иерархии будет следующим: cast-statue, statue, sculpture, casting, metalwork, standard-object, t. Иногда положение класса в таком списке называют его *специфичностью*. Классы расположены в нем по убыванию специфичности.

По списку предшествования можно определить, какой метод необходимо применить при вызове обобщенной функции. Этот процесс рассматривается в следующем разделе. Порядок следования, связанный с выбором наследуемого слота, нами пока не рассматривался. Соответствующие правила его определения можно найти на стр. 428.°

11.6. Обобщенные функции

Обобщенной функцией называют функцию, построенную из одного или нескольких методов. Методы определяются через оператор defmethod, который напоминает defun:

```
(defmethod combine (x y)
  (list x y))
```

На данный момент combine имеет один метод, и ее вызов построит список из двух аргументов:

```
> (combine 'a 'b)
(A B)
```

Пока что мы не сделали ничего, с чем не справилась бы обычная функция. Однако отличия станут заметны при добавлении еще одного метода.

Для начала создадим несколько классов, с которыми будут работать наши методы:

```
(defclass stuff () ((name :accessor name :initarg :name)))
(defclass ice-cream (stuff) ())
(defclass topping (stuff) ())
```

Мы определили три класса: stuff, имеющий слот name, а также ice-cream и topping, являющиеся подклассами stuff.

Теперь создадим второй метод для combine:

```
(defmethod combine ((ic ice-cream) (top topping))
  (format nil "~A ice-cream with ~A topping."
```

```
                (name ic)
                (name top)))
```

В этом вызове defmethod параметры *конкретизированы*: каждый из них появляется в списке с именем класса. Конкретизации метода указывают на типы аргументов, к которым он применим. Приведенный метод может использоваться только с определенными аргументами: первым аргументом должен быть экземпляр ice-cream, а вторым – экземпляр topping.

Как Лисп определяет, какой метод вызвать? Из тех, что удовлетворяют ограничениям на классы, будет выбран наиболее специфичный. Это значит, например, что для аргументов, представляющих классы ice-cream и topping соответственно, будет применен последний определенный нами метод:

```
> (combine (make-instance 'ice-cream :name 'fig)
           (make-instance 'topping :name 'treacle))
"FIG ice-cream with TREACLE topping."
```

Для любых других аргументов будет применен наш первый метод:

```
> (combine 23 'skiddoo)
(23 SKIDDOO)
```

Ни один из аргументов этого метода не был конкретизирован, поэтому он имеет наименьший приоритет и будет вызван лишь тогда, когда ни один другой метод не подойдет. Такие методы служат своего рода запасным вариантом, подобно ключу otherwise в case-выражении.

Любая комбинация параметров метода может быть конкретизирована. В следующем методе ограничивается лишь первый аргумент:

```
(defmethod combine ((ic ice-cream) x)
  (format nil "~A ice-cream with ~A."
          (name ic)
          x))
```

Если мы теперь вызовем combine с экземплярами ice-cream и topping, из двух последних методов будет выбран наиболее специфичный:

```
> (combine (make-instance 'ice-cream :name 'grape)
           (make-instance 'topping :name 'marshmallow))
"GRAPE ice-cream with MARSHMALLOW topping."
```

Однако если второй аргумент не принадлежит классу topping, будет вызван только что определенный метод:

```
> (combine (make-instance 'ice-cream :name 'clam)
           'reluctance)
"CLAM ice-cream with RELUCTANCE."
```

Аргументы обобщенной функции определяют набор *применимых* (*applicable*) методов. Метод считается применимым, если аргументы соответствуют спецификациям, налагаемым данным методом.

Отсутствие применимых методов приводит к ошибке. Если применим лишь один метод, он вызывается. Если применимо несколько методов, то в соответствии с порядком предшествования вызывается наиболее специфичный. Параметры сверяются слева направо. Если первый параметр одного из применимых методов ограничен более специфичным классом, чем первый аргумент других методов, то он и считается наиболее специфичным. Ничьи разрешаются переходом к следующему аргументу, и т. д.[1]

В предыдущих примерах легко найти наиболее специфичный применимый метод, так как все объекты можно расположить по убыванию: экземпляр `ice-cream` принадлежит классам `ice-cream`, затем `stuff`, `standard-object` и, наконец, `t`.

Методы не обязаны использовать спецификаторы, являющиеся только классами, определенными через `defclass`. Они также могут применять типы (точнее классы, соответствующие встроенным типам). Вот пример метода `combine` с числовыми спецификаторами:

```
(defmethod combine ((x number) (y number))
  (+ x y))
```

Кроме того, методы могут использовать отдельные объекты, сверяемые с помощью `eql`:

```
(defmethod combine ((x (eql 'powder)) (y (eql 'spark)))
  'boom)
```

Спецификаторы индивидуальных объектов имеют больший приоритет, нежели спецификаторы классов.

Списки аргументов методов могут быть довольно сложны, как и списки аргументов обычных функций, однако они должны быть *конгруэнтны* аргументам соответствующей обобщенной функции. Метод должен иметь столько же обязательных и необязательных параметров, сколько и обобщенная функция, и использовать либо `&rest`, либо `&key`, но не одновременно. Все приведенные ниже пары конгруэнтны:

```
(x)             (a)
(x &optional y) (a &optional b)
(x y &rest z)   (a b &key c)
(x y &key z)    (a b &key c d)
```

а следующие пары – нет:

```
(x)             (a b)
(x &optional y) (a &optional b c)
(x &optional y) (a &rest b)
(x &key x y)    (a)
```

[1] Мы не сможем дойти до конца аргументов и по-прежнему иметь ничью, потому что тогда два метода будут подходить точно под одно и то же описание аргументов. Это невозможно, так как второе определение метода в этом случае просто затрет первое.

Лишь обязательные параметры могут быть специализированы. Таким образом, метод полностью определяется именем и спецификациями обязательных параметров. Если мы определим еще один метод с тем же именем и спецификациями, он заменит предыдущий. То есть установив:

```
(defmethod combine ((x (eql 'powder)) (y (eql 'spark)))
  'kaboom)
```

мы тем самым переопределим поведение `combine` для аргументов `powder` и `spark`.

11.7. Вспомогательные методы

Методы могут дополняться *вспомогательными методами*, включая before- (перед), after- (после) и around-методы (вокруг). Before-методы позволяют нам сказать: «Прежде чем приступить к выполнению, сделайте это». Они вызываются в порядке убывания специфичности, предваряя вызов основного метода. After-методы позволяют сказать: «P.S. А сделайте заодно и это». Они вызываются после выполнения основного метода в порядке увеличения специфичности. То, что выполняется между ними, ранее называлось нами просто методом, но более точно это – *первичный метод*. В результате вызова возвращается именно его значение, даже если после него были вызваны after-методы.

Before- и after-методы позволяют добавлять новое поведение к вызову первичного метода. Around-методы позволяют выполнять то же самое, но более радикальным образом. Если задан around-метод, он будет вызван *вместо* первичного. Впрочем, по своему усмотрению он может вызвать первичный метод самостоятельно (с помощью функции `call-next-method`, которая предназначена именно для этой цели).

Этот механизм называется *стандартной комбинацией методов*. Согласно ему вызов обобщенной функции включает:

1. Вызов наиболее специфичного around-метода, если задан хотя бы один из них.
2. В противном случае по порядку:
   (a) Все before-методы в порядке убывания специфичности.
   (b) Наиболее специфичный первичный метод.
   (c) Все after-методы в порядке возрастания специфичности.

Возвращаемым значением считается значение around-метода (в случае 1) или значение наиболее специфичного первичного метода (в случае 2).

Вспомогательные методы задаются указанием ключа-квалификатора (qualifier) после имени метода в определении `defmethod`. Если мы определим первичный метод `speak` для класса `speaker` следующим образом:

```
(defclass speaker () ())

(defmethod speak ((s speaker) string)
```

```
  (format t "~A" string))
```

то вызов speak для экземпляра speaker просто напечатает второй аргумент:

```
> (speak (make-instance 'speaker)
         "I'm hungry")
I'm hungry
NIL
```

Определяя подкласс intellectual, оборачивающий before- и after-методы вокруг первичного метода speak:

```
(defclass intellectual (speaker) ())

(defmethod speak :before ((i intellectual) string)
  (princ "Perhaps "))

(defmethod speak :after ((i intellectual) string)
  (princ " in some sense"))
```

мы можем создать подкласс говорунов, всегда добавляющих в конец и в начало исходной фразы несколько слов:

```
> (speak (make-instance 'intellectual)
          "I'm hungry")
Perhaps I'm hungry in some sense
NIL
```

Как было упомянуто выше, вызываются все before- и after-методы. Если мы теперь определим before- и after-методы для суперкласса speaker:

```
(defmethod speak :before ((s speaker) string)
  (princ "I think "))
```

то они будут вызываться из середины нашего «сэндвича»:

```
> (speak (make-instance 'intellectual)
           "I'm hungry")
Perhaps I think I'm hungry in some sense
NIL
```

Независимо от того, вызываются ли before- и after-методы, при вызове обобщенной функции всегда возвращается значение первичного метода. В нашем случае format возвращает nil.

Это утверждение не распространяется на around-методы. Они вызываются сами по себе, а все остальные методы вызываются, лишь если им позволит around-метод. Around- или первичный методы могут вызывать следующий метод с помощью call-next-method. Перед этим уместно проверить наличие такого метода с помощью next-method-p.

С помощью around-методов вы можете определить другой, более предупредительный подкласс speaker:

```
(defclass courtier (speaker) ())
```

```
(defmethod speak :around ((c courtier) string)
  (format t "Does the King believe that ~A? " string)
  (if (eql (read) 'yes)
      (if (next-method-p) (call-next-method))
      (format t "Indeed, it is a preposterous idea.~%"))
  'bow)
```

Когда первым аргументом `speak` является экземпляр класса `courtier` (придворный), языком придворного управляет around-метод:

```
> (speak (make-instance 'courtier) "kings will last")
Does the King believe that kings will last? yes
I think kings will last
BOW
> (speak (make-instance 'courtier) "the world is round")
Does the King believe that the world is round? no
Indeed, it is a preposterous idea.
BOW
```

Еще раз обратите внимание, что, в отличие от before- и after-методов, значением вызова обобщенной функции является возвращаемое значение around-метода.

11.8. Комбинация методов

Единственный первичный метод, который будет вызываться в стандартной комбинации, является наиболее специфичным (при этом он может вызывать другие методы с помощью `call-next-method`). Но иногда нам хотелось бы иметь возможность комбинировать результаты всех подходящих по правилам выбора первичных методов.

Можно определить другие способы комбинации, например возврат суммы всех применимых методов. Под *операторной* комбинацией методов понимается случай, когда вызов завершается вычислением некоторого Лисп-выражения, являющегося применением оператора к результатам вызовов применимых первичных методов в порядке убывания их специфичности. Если мы определим обобщенную функцию `price`, комбинирующую значения с помощью функции `+`, и при этом для `price` не существует применимых around-методов, то она будет вести себя, как если бы она была определена следующим образом:

```
(defun price (&rest args)
  (+ (apply ⟨наиболее специфичный первичный метод⟩ args)
     .
     .
     .
     (apply ⟨наименее специфичный первичный метод⟩ args)))
```

Если существуют применимые around-методы, то они вызываются в порядке очередности, как в стандартной комбинации. В операторной комбинации around-метод по-прежнему может использовать `call-next-method`, но уже не может вызывать первичные методы.

Используемый способ комбинации методов может быть задан в момент явного создания обобщенной функции через defgeneric:

```
(defgeneric price (x)
  (:method-combination +))
```

Теперь метод price будет использовать + для комбинации методов; второй аргумент defmethod-определения должен содержать +. Определим некоторые классы с ценами:

```
(defclass jacket () ())
(defclass trousers () ())
(defclass suit (jacket trousers) ())

(defmethod price + ((jk jacket)) 350)
(defmethod price + ((tr trousers)) 200)
```

Теперь если мы запросим цену костюма (suit), то получим сумму применимых методов price:

```
> (price (make-instance 'suit))
550
```

Параметр :method-combination, передаваемый вторым аргументом defgeneric (а также вторым аргументом defmethod), может быть одним из следующих символов:

```
+   and   append   list   max   min   nconc   or   progn
```

Также можно использовать символ standard, который явным образом задает использование стандартной комбинации методов.

Определив единожды способ комбинации для обобщенной функции, вы вынуждены использовать этот способ для всех методов, имеющих то же имя. Попытка использовать другие символы (а также :before и :after) приведет к возникновению ошибки. Чтобы изменить способ комбинации price, вам придется удалить саму обобщенную функцию с помощью fmakunbound.

11.9. Инкапсуляция

Объектно-ориентированные языки часто дают возможность отличить внутреннее представление объекта от интерфейса, который они представляют внешнему миру. Сокрытие деталей реализации несет в себе двойную выгоду: вы можете модифицировать реализацию объекта, не затрагивая интерфейс доступа к нему извне, а также предотвращаете потенциально опасные изменения объектов. Такое сокрытие деталей иногда называют *инкапсуляцией*.

Несмотря на то, что инкапсуляцию часто считают свойством объектно-ориентированного программирования, эти две идеи в действительности существуют независимо друг от друга. Мы уже знакомились с ин-

капсуляцией в малом масштабе (стр. 120). Функции stamp и reset используют общую переменную counter, однако вызывающий код ничего о ней не знает и не может изменить ее напрямую.

В Common Lisp основным методом разделения информации на публичную и приватную являются пакеты. Чтобы ограничить доступ к чему-либо, мы кладем это в пакет, экспортируя из него лишь имена, являющиеся частью внешнего интерфейса.

Для инкапсуляции слота достаточно экспортировать лишь функции доступа к нему, но не имя самого слота. Например, мы можем определить класс counter и связать с ним методы increment и clear:

```
(defpackage "CTR"
            (:use "COMMON-LISP")
            (:export "COUNTER" "INCREMENT" "CLEAR"))

(in-package ctr)

(defclass counter () ((state :initform 0)))

(defmethod increment ((c counter))
  (incf (slot-value c 'state)))

(defmethod clear ((c counter))
  (setf (slot-value c 'state) 0))
```

Функции вне пакета ctr будут иметь возможность создавать экземпляры counter и вызывать increment и clear, однако не будут иметь доступа к самому слоту и имени state.

А если вы хотите не просто разделить внешний и внутренний интерфейсы класса, но и сделать *невозможным* сам доступ к значению слота, то можете поступить так: просто отинтернируйте (unintern) имя слота после выполнения кода, который ссылается на него:

```
(unintern 'state)
```

Теперь сослаться на слот невозможно ни из какого пакета.°

11.10. Две модели

Объектно-ориентированное программирование может сбить с толку отчасти из-за того, что есть две модели его реализации: модель передачи сообщений и модель обобщенных функций. Сначала появилась передача сообщений, а вторая модель по сути является обобщением первой.

В модели передачи сообщений методы принадлежат объектам и наследуются так же, как слоты. Чтобы найти площадь объекта, нужно послать ему сообщение area:

```
tell obj area
```

Это сообщение вызывает соответствующий метод, который `obj` определяет или наследует.

Иногда нам приходится сообщать дополнительные аргументы. Например, метод `move` может принимать аргумент, определяющий дальность перемещения. Чтобы переместить `obj` на `10`, пошлем ему следующее сообщение:

```
tell obj move 10
```

Если записать это другим способом:

```
(move obj 10)
```

то становится очевидной ограниченность модели передачи сообщений: мы можем специфицировать лишь первый аргумент. В такой модели не только не получится использовать методы для нескольких объектов, но и трудно даже представить себе такую возможность.

В модели передачи сообщений методы *принадлежат* объектам, в то время как в модели обобщенных функций методы специализируются *для* объектов. Если мы будем специализироваться только для первого аргумента, то эти две модели будут по сути одним и тем же. Но в модели обобщенных функций мы можем пойти дальше и специализироваться для любого необходимого количества аргументов. Таким образом, передача сообщений – лишь частный случай обобщенных функций, с помощью которых можно симулировать передачу сообщений, специализируя методы только по первому аргументу.

Итоги главы

1. В объектно-ориентированном программировании функция *f* определяется неявно при определении методов *f* для разных объектов. Объекты наследуют методы от своих родителей.
2. Определение класса напоминает более многословное определение структуры. Разделяемый слот принадлежит всему классу.
3. Класс наследует слоты своих суперклассов.
4. Предки класса упорядочены в соответствии со списком предшествования. Алгоритм предшествования легче всего понять визуально.
5. Обобщенная функция состоит из всех методов с тем же именем. Метод определяется именем и специализацией своих параметров. Метод, используемый при вызове обобщенной функции, определяется согласно порядку предшествования.
6. Первичные методы могут быть дополнены вспомогательными. Стандартная комбинация методов означает использование around-методов, если таковые имеются; в противном случае вызываются сначала before-, потом первичный, затем after-методы.
7. В операторной комбинации методов все первичные методы рассматриваются как аргументы заданного оператора.

8. Инкапсуляция может быть осуществлена с помощью пакетов.
9. Существуют две модели объектно-ориентированного программирования. Модель обобщенных функций является обобщением модели передачи сообщений.

Упражнения

1. Определите параметры `accessor`, `initform`, `initarg` для классов, определенных на рис. 11.2. Внесите необходимые правки, чтобы код более не использовал `slot-value`.
2. Перепишите код на рис. 9.5 так, чтобы сферы и точки были классами, а `introspect` и `normal` – обобщенными функциями.
3. Имеется набор классов:

```
(defclass a (c d) ...)      (defclass e () ...)
(defclass b (d c) ...)      (defclass f (h) ...)
(defclass c () ...)         (defclass g (h) ...)
(defclass d (e f g) ...)    (defclass h () ...)
```

   (a) Нарисуйте граф, представляющий предков `a`, и составьте список классов, которым принадлежит экземпляр `a` в порядке убывания их специфичности.

   (b) Повторите аналогичную процедуру для `b`.
4. Имеются следующие функции:
   - `precedence`: принимает объект и возвращает для него список предшествования классов по убыванию специфичности.
   - `methods`: принимает обобщенную функцию и возвращает список всех ее методов.
   - `specializations`: принимает метод и возвращает список специализаций его параметров. Каждый элемент возвращаемого списка будет либо классом, либо списком вида `(eql` *x*`)`, либо `t` (для неспециализированного параметра).

   С помощью перечисленных функций (не пользуясь `compute-applicable-methods` или `find-method`) определите функцию `most-spec-app-meth`, принимающую обобщенную функцию и список аргументов, с которыми она будет вызвана, и возвращающую наиболее специфичный применимый метод, если таковые имеются.
5. Измените код на рис. 11.2 так, чтобы при каждом вызове обобщенной функции `area` увеличивался некий глобальный счетчик. Функция не должна менять свое поведение никаким иным образом.
6. Приведите пример задачи, трудноразрешимой в том случае, когда лишь первый аргумент обобщенной функции может быть специализирован.

12

Структура

В разделе 3.3 было объяснено, каким образом использование указателей позволяет поместить любое значение куда угодно. Это утверждение включает массу возможностей, но не все они могут принести пользу. Например, объект может быть элементом самого себя. Хорошо это или плохо, зависит от того, делается это намеренно или происходит случайно.

12.1. Разделяемая структура

Списки могут совместно использовать одни и те же ячейки. В простейшем случае один список может быть частью другого. После выполнения

```
> (setf part (list 'b 'c))
(B C)
> (setf whole (cons 'a part))
(A B C)
```

первая ячейка становится частью (а точнее, `cdr`) второй. В подобных случаях принято говорить, что два списка *разделяют одну структуру*. Структура, лежащая в основе двух таких списков, представлена на рис. 12.1.

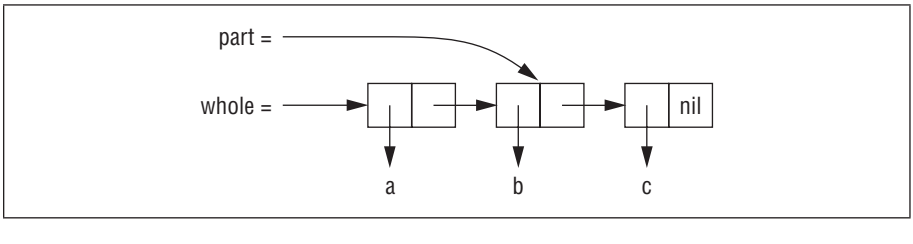

Рис. 12.1. *Разделяемая структура*

Подобные ситуации выявляет предикат `tailp`. Он принимает два списка и возвращает истину, если встретит первый список при обходе второго.

```
> (tailp part whole)
T
```

Мы можем реализовать его самостоятельно:

```
(defun our-tailp (x y)
  (or (eql x y)
      (and (consp y)
           (our-tailp x (cdr y)))))
```

Согласно этому определению каждый список является хвостом самого себя, а `nil` является хвостом любого правильного списка.

В более сложном случае два списка могут разделять общую структуру, даже когда один из них не является хвостом другого. Это происходит, когда они делят общий хвост, как показано на рис. 12.2. Создадим подобную ситуацию:

```
(setf part (list 'b 'c)
      whole1 (cons 1 part)
      whole2 (cons 2 part))
```

Теперь `whole1` и `whole2` разделяют структуру, но один не является хвостом другого.

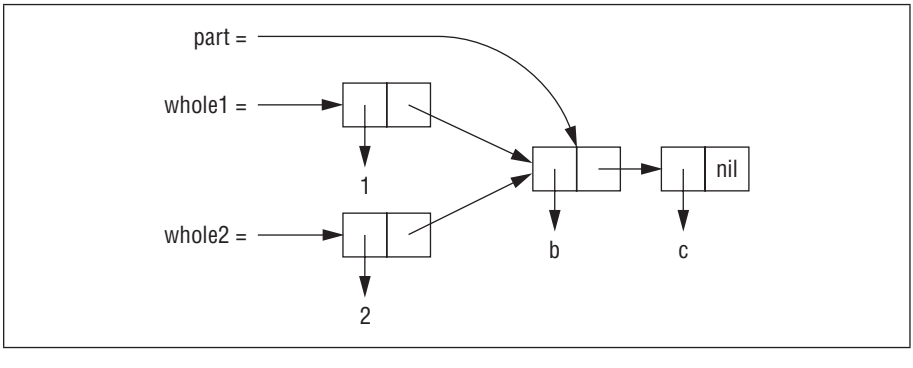

Рис. 12.2. *Разделяемый хвост*

В случае вложенных списков важно отличать списки с разделяемой структурой от их элементов с разделяемой структурой. *Структура списка верхнего уровня* включает ячейки, из которых состоит сам список, но не включает какие-либо ячейки, из которых состоят отдельные элементы списка. Пример структуры верхнего уровня вложенного списка приведен на рис. 12.3.

Имеют ли две ячейки разделяемую структуру, зависит от того, считаем мы их списками или деревьями. Два вложенных списка могут разделять одну структуру как деревья, но не разделять ее как списки. Сле-

дующий код создает ситуацию, изображенную на рис. 12.4, где два списка содержат один и тот же элемент-список:

```
(setf element (list 'a 'b)
      holds1  (list 1 element 2)
      holds2  (list element 3))
```

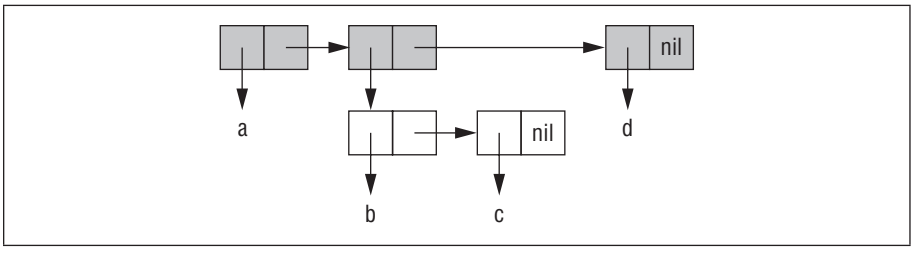

Рис. 12.3. *Структура списка верхнего уровня*

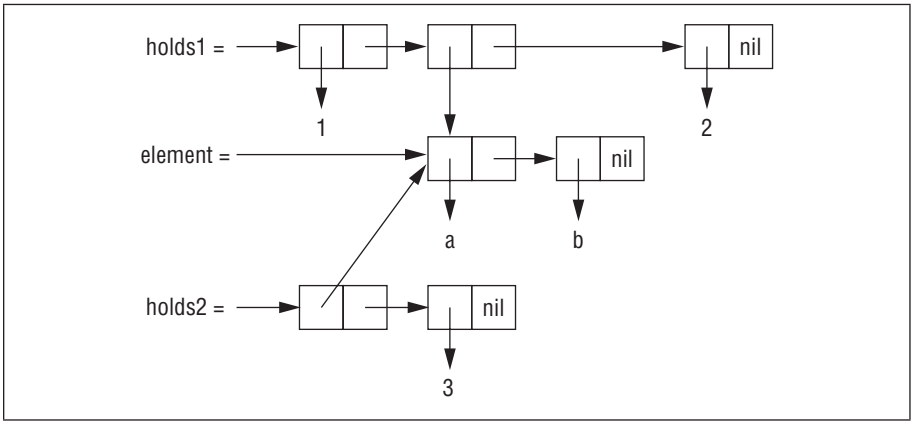

Рис. 12.4. *Разделяемое поддерево*

Хотя второй элемент `holds1` разделяет структуру с первым элементом `holds2` (в действительности, он ему идентичен), `holds1` и `holds2` не делят между собой общую структуру как списки. Два списка разделяют структуру *как списки*, только если они делят общую структуру верхнего уровня, чего не делают `holds1` и `holds2`.

Избежать использования разделяемой структуры можно с помощью копирования. Функция `copy-list`, определяемая как

```
(defun our-copy-list (lst)
  (if (null lst)
      nil
      (cons (car lst) (our-copy-list (cdr lst)))))
```

вернет список, не разделяющий структуру верхнего уровня с исходным списком. Функция copy-tree, которая может быть определена следующим образом:

```
(defun our-copy-tree (tr)
  (if (atom tr)
      tr
      (cons (our-copy-tree (car tr))
            (our-copy-tree (cdr tr)))))
```

возвратит список, который не разделяет структуру всего дерева с исходным списком. Рисунок 12.5 демонстрирует разницу между вызовом copy-list и copy-tree для вложенного списка.

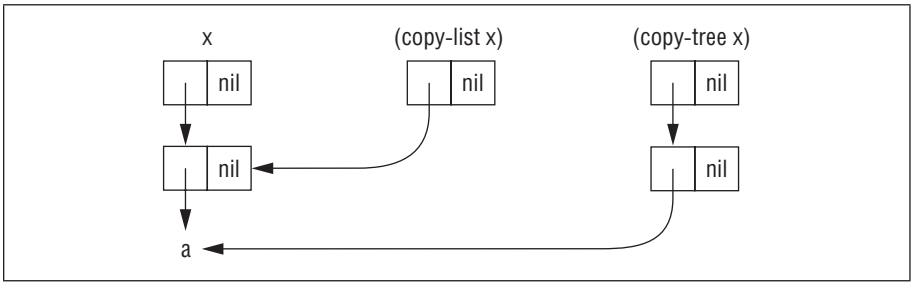

Рис. 12.5. *Два способа копирования*

12.2. Модификация

Почему стоит избегать использования разделяемой структуры? До сих пор это явление рассматривалось всего лишь как забавная головоломка, а в написанных ранее программах мы прекрасно обходились и без него. Разделяемая структура вызывает проблемы, если объекты, которые имеют такую структуру, модифицируются. Дело в том, что модификация одного из двух списков с общей структурой повлечет за собой непреднамеренное изменение другого.

В предыдущем разделе мы видели, как сделать один список хвостом другого:

```
(setf whole (list 'a 'b 'c)
      tail (cdr whole))
```

Изменение списка tail повлечет симметричное изменение хвоста whole, и наоборот, так как по сути это одна и та же ячейка:

```
> (setf (second tail) 'e)
E
> tail
(B E)
```

```
> whole
(A B E)
```

Разумеется, то же самое будет происходить и для двух списков, имеющих общий хвост.

Изменение двух объектов одновременно не всегда является ошибкой. Иногда это именно то, что нужно. Однако если такое изменение происходит непреднамеренно, оно может привести к некорректной работе программы. Опытные программисты умеют избегать подобных ошибок и немедленно распознавать такие ситуации. Если список без видимой причины меняет свое содержимое, вероятно, он имеет разделяемую структуру.

Опасна не сама разделяемая структура, а возможность ее изменения. Чтобы гарантировать отсутствие подобных ошибок, просто избегайте использования `setf` (а также аналогичных операторов типа `pop`, `rplaca` и других) для списков. Если изменяемость списков все же требуется, то необходимо выяснить, откуда взялся изменяемый список, чтобы убедиться, что он не делит структуру с чем-то, что нельзя менять. Если же это не так или вам неизвестно происхождение списка, то модифицировать необходимо не сам список, а его копию.

Нужно быть вдвойне осторожным при использовании функций, написанных кем-то другим. Пока не установлено обратное, имейте в виду, что все, что передается функции:

1. Может быть передано деструктивным операторам.
2. Может быть сохранено где-либо, и изменение этого объекта приведет к изменению в других частях кода, использующего данный объект.[1]

В обоих случаях правильным решением является копирование аргументов.

В Common Lisp вызов любой функции, выполняющейся во время прохода по структуре (например, функции-аргумента `mapcar` или `remove-if`), не должен менять эту структуру. В противном случае последствия выполнения такого кода не определены.

12.3. Пример: очереди

Разделяемые структуры – это не только повод для беспокойства. Иногда они могут быть полезны. В этом разделе показано, как с помощью разделяемых структур представить очереди. Очередь – это хранилище объектов, из которого они могут быть извлечены по одному в том же

[1] Например, в Common Lisp попытка изменения строки, используемой как имя символа, считается ошибкой. И поскольку нам неизвестно, копирует ли `intern` свой аргумент-строку перед созданием символа, мы должны предположить, что модификация любой строки, которая передавалась в `intern`, приведет к ошибке.

порядке, в котором они были туда записаны. Такую модель принято именовать FIFO – сокращение от «первым пришел, первым ушел» («first in, first out»).

С помощью списков легко представить стопку, так как добавление и получение элементов происходит с одного конца. Задача представления очереди более сложна, поскольку добавление и извлечение объектов происходит с разных концов. Для эффективной ее реализации необходимо каким-то образом обеспечить управление обоими концами списка.

Одна из возможных стратегий приводится на рис. 12.6, где показана очередь из трех элементов: `a`, `b` и `c`. Очередью считаем точечную пару, состоящую из списка и последней ячейки этого же списка. Будем называть их *начало* и *конец*. Чтобы получить элемент из очереди, необходимо просто извлечь *начало*. Чтобы добавить новый элемент, необходимо создать новую ячейку, сделать ее `cdr` конца очереди и затем сделать ее же *концом*.

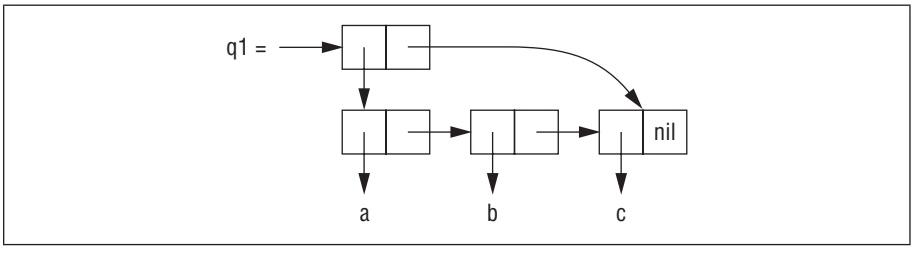

Рис. 12.6. *Структура очереди*

Такую стратегию реализует код на рис. 12.7.

```
(defun make-queue () (cons nil nil))

(defun enqueue (obj q)
  (if (null (car q))
      (setf (cdr q) (setf (car q) (list obj)))
      (setf (cdr (cdr q)) (list obj)
            (cdr q) (cdr (cdr q))))
  (car q))

(defun dequeue (q)
  (pop (car q)))
```

Рис. 12.7. *Реализация очередей*

Она используется следующим образом:

```
> (setf q1 (make-queue))
(NIL)
```

```
> (progn (enqueue 'a q1)
         (enqueue 'b q1)
         (enqueue 'c q1))
(A B C)
```

Теперь q1 представляет очередь, изображенную на рис. 12.6:

```
> q1
((A B C) C)
```

Попробуем забрать из очереди несколько элементов:

```
> (dequeue q1)
A
> (dequeue q1)
B
> (enqueue 'd q1)
(C D)
```

12.4. Деструктивные функции

Common Lisp включает в себя несколько функций, которые могут изменять структуру списков и за счет этого работать быстрее. Они называются деструктивными. И хотя эти функции могут изменять ячейки, переданные им в качестве аргументов, они делают это не ради побочных эффектов.

Например, delete является деструктивным аналогом remove. Хотя ей и позволено портить переданный список, она не дает никаких обещаний, что так и будет делать. Посмотрим, что происходит в большинстве реализаций:

```
> (setf lst '(a r a b i a))
(A R A B I A)
> (delete 'a lst)
(R B I)
> lst
(A R B I)
```

Как и в случае с remove, чтобы зафиксировать побочный эффект, необходимо использовать setf:

```
(setf lst (delete 'a lst))
```

Примером того, как деструктивные функции модифицируют списки, является nconc, деструктивная версия append.[1] Приведем ее версию для двух аргументов, демонстрирующую, каким образом сшиваются списки:

```
(defun nconc2 (x y)
  (if (consp x)
```

[1] Буква n в имени nconc соответствует «non-consing». Имена многих деструктивных функций начинаются с n.

```
        (progn
          (setf (cdr (last x)) y)
          x)
        y))
```

cdr последней ячейки первого списка становится указателем на второй список. Полноценная версия для произвольного числа аргументов приводится в приложении B.

Функция mapcan похожа на mapcar, но соединяет в один список возвращаемые значения (которые должны быть списками) с помощью nconc:

```
> (mapcan #'list
          '(a b c)
          '(1 2 3 4))
(A 1 B 2 C 3)
```

Эта функция может быть определена следующим образом:

```
(defun our-mapcan (fn &rest lsts)
  (apply #'nconc (apply #'mapcar fn lsts)))
```

Используйте mapcan с осторожностью, учитывая ее деструктивный характер. Она соединяет возвращаемые списки с помощью nconc, поэтому их лучше больше нигде не задействовать.

Функция mapcan полезна, в частности, в задачах, интерпретируемых как сбор всех узлов одного уровня некоего дерева. Например, если children возвращает список чьих-то детей, тогда мы сможем определить функцию для получения списка внуков так:

```
(defun grandchildren (x)
  (mapcan #'(lambda (c)
              (copy-list (children c)))
          (children x)))
```

Данная функция применяет copy-list к результату вызова children, так как он может возвращать уже существующий объект, а не производить новый.

Также можно определить недеструктивный вариант mapcan:

```
(defun mappend (fn &rest lsts)
  (apply #'append (apply #'mapcar fn lsts)))
```

Используя mappend, мы можем обойтись без вызовов copy-list в определении grandchildren:

```
(defun grandchildren (x)
  (mappend #'children (children x)))
```

12.5. Пример: двоичные деревья поиска

В некоторых ситуациях уместнее использовать деструктивные операции, нежели недеструктивные. В разделе 4.7 было показано, как управ-

лять двоичными деревьями поиска (BST). Все использованные там функции были недеструктивными, но если нужно применить BST на практике, такая осторожность излишня.

На рис. 12.8 приведена деструктивная версия bst-insert (стр. 86). Она принимает точно такие же аргументы и возвращает точно такое же значение, как и исходная версия. Единственным отличием является то, что она может изменять дерево, передаваемое вторым аргументом.

```
(defun bst-insert! (obj bst <)
  (if (null bst)
      (make-node :elt obj)
      (progn (bsti obj bst <)
             bst)))

(defun bsti (obj bst <)
  (let ((elt (node-elt bst)))
    (if (eql obj elt)
        bst
        (if (funcall < obj elt)
            (let ((l (node-l bst)))
              (if l
                  (bsti obj l <)
                  (setf (node-l bst)
                        (make-node :elt obj))))
            (let ((r (node-r bst)))
              (if r
                  (bsti obj r <)
                  (setf (node-r bst)
                        (make-node :elt obj))))))))
```

Рис. 12.8. *Двоичные деревья поиска: деструктивная вставка*

В разделе 2.12 было предупреждение о том, что деструктивные функции вызываются не ради побочных эффектов. Поэтому если вы хотите построить дерево с помощью bst-insert!, вам нужно вызывать ее так же, как если бы вы вызывали настоящую bst-insert:

```
> (setf *bst* nil)
NIL
> (dolist (x '(7 2 9 8 4 1 5 12))
    (setf *bst* (bst-insert! x *bst* #'<)))
NIL
```

Вы *могли бы* также определить аналог push для BST, но это довольно сложно и выходит за рамки данной книги. (Для любопытных такой макрос определяется на стр. 429.°)

На рис. 12.9[1] представлен деструктивный вариант функции bst-delete, которая связана с bst-remove (стр. 89) так же, как delete связана с remove. Как и delete, она не подразумевает вызов ради побочных эффектов. Использовать bst-delete необходимо так же, как и bst-remove:

```
> (setf *bst* (bst-delete 2 *bst* #'<))
#<7>
> (bst-find 2 *bst* #'<)
NIL
```

```
(defun bst-delete (obj bst <)
  (if (null bst)
      nil
      (if (eql obj (node-elt bst))
          (del-root bst)
          (progn
            (if (funcall < obj (node-elt bst))
                (setf (node-l bst) (bst-delete obj (node-l bst) <))
                (setf (node-r bst) (bst-delete obj (node-r bst) <)))
            bst))))

(defun del-root (bst)
  (let ((l (node-l bst)) (r (node-r bst)))
    (cond ((null l) r)
          ((null r) l)
          (t        (if (zerop (random 2))
                       (cutnext r bst nil)
                        (cutprev l bst nil))))))

(defun cutnext (bst root prev)
  (if (node-l bst)
      (cutnext (node-l bst) root bst)
      (if prev
          (progn
            (setf (node-elt root) (node-elt bst)
                  (node-l prev)   (node-r bst))
            root)
          (progn
            (setf (node-l bst)    (node-l root))
            bst))))

(defun cutprev (bst root prev)
  (if (node-r bst)
      (cutprev (node-r bst) root bst)
      (if prev
          (progn
            (setf (node-elt root) (node-elt bst)
```

[1] Данный листинг содержит исправленную версию bst-delete. За подробностями обращайтесь к сноске на стр. 89. – *Прим. перев.*

```
                 (node-r prev)   (node-l bst))
            root)
          (progn
            (setf (node-r bst)    (node-r root))
            bst))))

(defun replace-node (old new)
  (setf (node-elt old) (node-elt new)
        (node-l   old) (node-l   new)
        (node-r   old) (node-r   new)))

(defun cutmin (bst par dir)
  (if (node-l bst)
      (cutmin (node-l bst) bst :l)
      (progn
        (set-par par dir (node-r bst))
        (node-elt bst))))

(defun cutmax (bst par dir)
  (if (node-r bst)
      (cutmax (node-r bst) bst :r)
      (progn
        (set-par par dir (node-l bst))
        (node-elt bst))))

(defun set-par (par dir val)
  (case dir
    (:l (setf (node-l par) val))
    (:r (setf (node-r par) val))))
```

Рис. 12.9. *Двоичные деревья поиска: деструктивное удаление*

12.6. Пример: двусвязные списки

Обычные списки в Лиспе являются односвязными. Это означает, что движение по указателям происходит только в одном направлении: вы можете перейти к следующему элементу, но не можете вернуться к предыдущему. *Двусвязные списки* имеют также и обратный указатель, поэтому можно перемещаться в обе стороны. В этом разделе показано, как создавать и использовать двусвязные списки.

На рис. 12.10 показана их возможная реализация. `cons`-ячейки имеют два поля: `car`, указывающий на данные, и `cdr`, указывающий на следующий элемент. Элемент двусвязного списка должен иметь еще одно поле, указывающее на предыдущий элемент. Вызов `defstruct` (рис. 12.10) создает объект из трех частей, названный `dl` (от «doubly linked»), который мы будем использовать для создания двусвязных списков. Поле `data` в `dl` соответствует `car` в `cons`-ячейке, а поле `next` соответствует `cdr`. Поле

prev похоже на cdr, но указывает в обратном направлении. (Пример такой структуры приведен на рис. 12.11.) Пустому двусвязному списку, как и обычному, соответствует nil.

Вызов defstruct также определяет функции для двусвязных списков, аналогичные car, cdr и consp: dl-data, dl-next и dl-p. Функция печати dl->list возвращает обычный список с теми же значениями, что и двусвязный.

Функция dl-insert похожа на cons. По крайней мере, она, как и cons, является основной функцией-конструктором. В отличие от cons, она изменяет двусвязный список, переданный вторым аргументом. В данной ситуации это совершенно нормально. Чтобы поместить новый объект в начало обычного списка, вам не требуется его изменять, однако чтобы поместить объект в начало двусвязного списка, необходимо присвоить полю prev указатель на новый объект.

```
(defstruct (dl (:print-function print-dl))
  prev data next)

(defun print-dl (dl stream depth)
  (declare (ignore depth))
  (format stream "#<DL ~A>" (dl->list dl)))

(defun dl->list (lst)
  (if (dl-p lst)
      (cons (dl-data lst) (dl->list (dl-next lst)))
      lst))

(defun dl-insert (x lst)
  (let ((elt (make-dl :data x :next lst)))
    (when (dl-p lst)
      (if (dl-prev lst)
          (setf (dl-next (dl-prev lst)) elt
                (dl-prev elt) (dl-prev lst)))
      (setf (dl-prev lst) elt))
    elt))

(defun dl-list (&rest args)
  (reduce #'dl-insert args
          :from-end t :initial-value nil))

(defun dl-remove (lst)
  (if (dl-prev lst)
      (setf (dl-next (dl-prev lst)) (dl-next lst)))
  (if (dl-next lst)
      (setf (dl-prev (dl-next lst)) (dl-prev lst)))
  (dl-next lst))
```

Рис. 12.10. *Построение двусвязных списков*

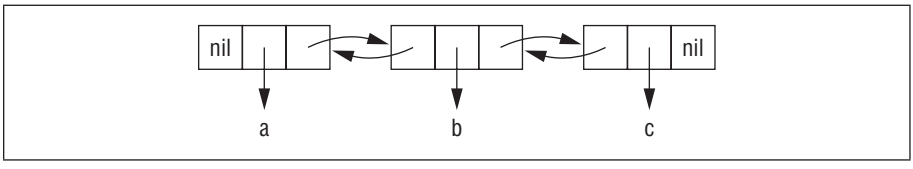

Рис. 12.11. *Двусвязный список*

Другими словами, несколько обычных списков могут иметь общий хвост. Но для пары двусвязных списков это невозможно, так как хвост каждого из них имеет разные указатели на голову. Если бы функция `dl-insert` не была деструктивна, ей бы приходилось всегда копировать свой второй аргумент.

Другое интересное различие между одно- и двусвязными списками заключается в способе доступа к их элементам. Работая с односвязным списком, вы храните указатель на его начало. При работе с двусвязным списком, поскольку в нем элементы соединены с обоих концов, вы можете использовать указатель на любой из элементов. Поэтому `dl-insert`, в отличие от `cons`, может добавлять новый элемент в любое место двусвязного списка, а не только в начало.

Функция `dl-list` является `dl`-аналогом `list`. Она получает произвольное количество аргументов и возвращает состоящий из них `dl`:

```
> (dl-list 'a 'b 'c)
#<DL (A B C)>
```

В ней используется `reduce` с параметрами: `:from-end`, установленным в `t`, и `:initial-value`, установленным в `nil`, что делает приведенный выше вызов эквивалентным следующей последовательности:

```
(dl-insert 'a (dl-insert 'b (dl-insert 'c nil)))
```

Заменив `#'dl-insert` на `#'cons` в определении `dl-list`, эта функция будет вести себя аналогично `list`:

```
> (setf dl (dl-list 'a 'b))
#<DL (A B)>
> (setf dl (dl-insert 'c dl))
#<DL (C A B)>
> (dl-insert 'r (dl-next dl))
#<DL (R A B)>
> dl
#<DL (C R A B)>
```

Наконец, для удаления элемента из двусвязного списка определена `dl-remove`. Как и `dl-insert`, она сделана деструктивной.

12.7. Циклическая структура

Изменяя структуру списков, можно создавать циклические списки. Они бывают двух видов. Наиболее полезными являются те, которые имеют замкнутую структуру верхнего уровня. Такие списки называются *циклическими по хвосту* (*cdr-circular*), так как цикл создается `cdr`-частями ячеек.

Чтобы создать такой список, содержащий один элемент, необходимо установить указатель `cdr` на самого себя:

```
> (setf x (list 'a))
(A)
> (progn (setf (cdr x) x) nil)
NIL
```

Теперь `x` – циклический список. Его структура изображена на рис. 12.12.

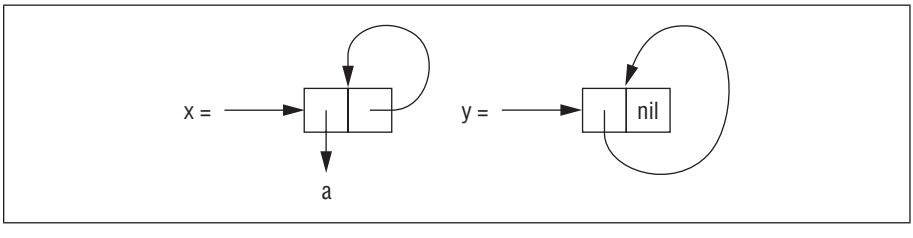

Рис. 12.12. *Циклические списки*

При попытке напечатать такой список символ `a` будет выводиться до бесконечности. Этого можно избежать, установив значение `*print-circle*` в `t`:

```
> (setf *print-circle* t)
T
> x
#1=(A . #1#)
```

При необходимости можно использовать макросы чтения `#n=` и `#n#` для представления такой структуры.

Списки с циклическим хвостом могут быть полезны для представления, например, буферов или ограниченных наборов каких-то объектов (пулов[1]). Следующая функция превратит произвольный нециклический непустой список в циклический с теми же элементами:

```
(defun circular (lst)
  (setf (cdr (last lst)) lst))
```

[1] Пул – это набор инициализированных ресурсов, которые поддерживаются в готовом к использованию состоянии, а не выделяются по требованию. – *Прим. ред.*

Другой тип циклических списков – *циклические по голове* (*car-circular*). Список такого типа можно понимать как дерево, являющееся поддеревом самого себя. Его название обусловлено тем, что в нем содержится цикл, замкнутый на car ячейки. Ниже мы создадим циклический по голове список, второй элемент которого является им самим:

```
> (let ((y (list 'a)))
    (setf (car y) y)
    y)
#1=(#1#)
```

Результат изображен на рис. 12.12. Несмотря на цикличность, этот циклический по голове список (car-circular) по-прежнему является правильным списком, в отличие от циклических по хвосту (cdr-circular), которые правильными быть не могут.

Список может быть циклическим по голове и хвосту одновременно. car и cdr такой ячейки будут указывать на нее саму:

```
> (let ((c (cons 1 1)))
    (setf (car c) c
          (cdr c) c)
    c)
#1=(#1# . #1#)
```

Сложно представить, для чего могут использоваться подобные объекты. На самом деле, главное, что нужно вынести из этого, – необходимо избегать непреднамеренного создания циклических списков, так как большинство функций, работающих со списками, будут уходить в бесконечный цикл, если получат в качестве аргумента список, циклический по тому направлению, по которому они совершают проход.

Циклическая структура может быть проблемой не только для списков, но и для других типов объектов, например для массивов:

```
> (setf *print-array* t)
T
> (let ((a (make-array 1)))
    (setf (aref a 0) a)
    a)
#1=#(#1#)
```

И действительно, практически любой объект, состоящий из элементов, может включать себя в качестве одного из них.

Разумеется, структуры, создаваемые defstruct, также могут быть циклическими. Например, структура *c*, представляющая элемент дерева, может иметь поле parent, содержащее другую структуру *p*, чье поле child ссылается обратно на *c*:

```
> (progn (defstruct elt
           (parent nil) (child nil))
         (let ((c (make-elt))
               (p (make-elt)))
```

```
            (setf (elt-parent c) p
                  (elt-child  p) c)
            c))
#1=#S(ELT PARENT #S(ELT PARENT NIL CHILD #1#) CHILD NIL)
```

Для печати подобных структур необходимо установить `*print-circle*` в `t` или же избегать вывода таких объектов.

12.8. Неизменяемая структура

Неизменяемые объекты также являются частью кода, и наша задача не допускать их модификации, потому что в противном случае мы случайно можем создать программу, которая пишет сама себя. Цитируемый список является константой, поэтому следует проявлять аккуратность при работе с отдельными его ячейками. Например, если мы используем следующий предикат для проверки на принадлежность к арифметическим операторам,

```
(defun arith-op (x)
  (member x '(+ - * /)))
```

то в случае истинного значения он будет возвращать часть цитируемого списка. Изменяя возвращаемое значение:

```
> (nconc (arith-op '*) '(as it were))
(* / AS IT WERE)
```

мы изменяем и сам исходный список внутри `arith-op`, что приведет к изменению работы этой функции:

```
> (arith-op 'as)
(AS IT WERE)
```

Возврат константы из функции не всегда является ошибкой. Однако нужно учитывать подобные случаи при принятии решения о безопасности выполнения деструктивных операций над чем-либо.

Избежать возврата части неизменной структуры в случае `arith-op` можно несколькими способами. В общем случае замена цитирования на явный вызов функции `list` решит проблему и приведет к созданию нового списка при каждом вызове:

```
(defun arith-op (x)
  (member x (list '+ '- '* '/)))
```

Однако в данном случае вызов `list` ударит по производительности, поэтому здесь лучше вместо `member` использовать `find`:

```
(defun arith-op (x)
  (find x '(+ - * /)))
```

Проблема, рассмотренная в этом разделе, чаще всего возникает при работе со списками, однако она актуальна и для других типов данных:

массивов, строк, структур, экземпляров и т. д. Не следует модифицировать что-либо, заданное в тексте программы буквально.

Даже если вы собираетесь написать самомодифицируемую программу, изменение структур-констант – это неправильный выбор. Компилятор *может* связывать константы с кодом, а деструктивные операторы *могут* изменять свои аргументы, но ни то, ни другое не гарантируется. Писать самомодифицируемые программы вы можете, например, с помощью замыканий (см. раздел 6.5).

Итоги главы

1. Два списка могут разделять общий хвост. Списки могут разделять структуры как деревья, без разделения структуры верхнего уровня. Разделения структур можно избежать с помощью копирования.
2. Разделяемость структуры не влияет на поведение функций, но о ней нельзя забывать, если вы собираетесь модифицировать списки. Если два списка разделяют общую структуру, то изменение одного из них может привести к изменению другого.
3. Очереди могут быть представлены как `cons`-ячейки, `car` которых указывает на первую ячейку списка, а `cdr` – на последнюю.
4. Из соображений производительности деструктивным операторам разрешается модифицировать свои аргументы.
5. В некоторых приложениях использование деструктивных операторов более естественно.
6. Списки могут быть циклическими по голове и хвосту. Лисп умеет работать с циклическими и разделяемыми структурами.
7. Не следует изменять константы, встречающиеся в тексте программы.

Упражнения

1. Нарисуйте три различных дерева, которые будут выводиться как `((A) (A) (A))`. Напишите выражения, генерирующие каждое из них.
2. Считая уже определенными `make-queue`, `enqueue` и `dequeue` (см. рис. 12.7), нарисуйте представление очереди в виде ячеек после каждого следующего шага:

```
> (setf q (make-queue))
(NIL)
> (enqueue 'a q)
(A)
> (enqueu 'b q)
(A B)
> (dequeue q)
A
```

3. Определите функцию `copy-queue`, возвращающую копию очереди.
4. Определите функцию, принимающую в качестве аргументов объект и очередь и помещающую этот объект в *начало* очереди.
5. Определите функцию, принимающую в качестве аргументов объект и очередь и (деструктивно) перемещающую первый найденный (`eql`) экземпляр этого объекта в начало очереди.
6. Определите функцию, принимающую объект и циклический список, который может быть циклическим по хвосту, и проверяющую наличие заданного объекта в списке.
7. Определите функцию, проверяющую, является ли ее аргумент циклическим по хвосту списком.
8. Определите функцию, проверяющую, является ли ее аргумент циклическим по голове списком.

13

Скорость

На самом деле, Лисп сочетает в себе два языка: язык для быстрого написания программ и язык для написания быстрых программ. На первых этапах создания программы вы можете пожертвовать ее скоростью ради удобства разработки, а когда ее структура примет окончательную форму, можно воспользоваться инструментами языка, позволяющими увеличить скорость работы программы.

Довольно сложно давать общие рекомендации по поводу оптимизации из-за внушительного разнообразия реализаций Common Lisp. Действие, ускоряющее выполнение вашей программы в одной из них, может замедлить работу в другой. Чем мощнее язык, тем дальше он отстоит от аппаратной части, а чем дальше вы от аппаратной части, тем выше шансы, что разные реализации по-разному будут возвращаться к ней при генерации программного кода. Поэтому, несмотря на то что некоторые методики, описанные ниже, почти наверняка ускорят выполнение кода в любой реализации, все же не стоит забывать о рекомендательном характере содержимого этой главы.

13.1. Правило бутылочного горлышка

Независимо от реализации есть три момента, касающиеся оптимизации: она должна быть сосредоточена на бутылочных горлышках, она не должна начинаться слишком рано и она должна начинаться с алгоритмов. Пожалуй, по поводу оптимизации важнее всего понять, что в любой программе, как правило, есть несколько узких мест, выполнение которых занимает большую часть времени. По мнению Кнута, «подавляющая часть времени выполнения программы, не связанной с вводом-выводом, сосредоточена в примерно 3% исходного кода».° Оптимизация таких участков даст существенно большее увеличение скорости, чем оптимизация остальных частей, которая будет пустой тратой времени.

Поэтому исключительно важный первый шаг оптимизации любой программы – поиск подобных узких мест, или бутылочных горлышек. Многие реализации Лиспа включают собственные *профилировщики*, которые наблюдают за выполнением программы и предоставляют отчет о времени, затраченном на каждую из ее частей, выявляя тем самым узкие места. Профилировщик – ценный инструмент, совершенно необходимый для создания действительно эффективного кода. Если ваша реализация Лиспа включает профилировщик, то наверняка предоставляет и документацию на него, которую следует изучить. Если же такого инструмента у вас нет, то узкие места придется угадывать самостоятельно, и вы будете удивлены, когда узнаете, как часто подобные догадки бывают ошибочными.

Следствие правила бутылочных горлышек: не стоит вкладывать слишком много усилий в оптимизацию на ранних стадиях написания программы. Кнут предлагает еще более жесткую формулировку: «Преждевременная оптимизация является корнем всех зол (или, по крайней мере, большей части) в программировании».° Пока программа не написана, сложно сказать, где возникнет узкое место, поэтому не тратьте свое время зря. Кроме того, оптимизация затрудняет модификацию, и попытка оптимизировать программу в момент ее написания сродни попытке писать картину быстросохнущими красками.

Вероятность того, что программа получится хорошей, возрастет, если на каждой подзадаче можно будет сфокусироваться в подходящий момент времени. Одним из достоинств Лиспа является возможность работать в разном ритме: быстро писать медленный код или медленно писать быстрый код. На первых этапах вы используете первый метод, а на стадии оптимизации переключаетесь на второй. Эта модель соответствует правилу бутылочного горлышка. В низкоуровневых языках, таких как ассемблер, вы фактически оптимизируете каждую строку кода. Большая часть этих усилий тратится напрасно, так как бутылочные горлышки составляют лишь небольшую часть всего кода. Более абстрактный язык позволяет уделить узким местам значительную долю времени, поэтому при меньших усилиях вы получаете существенно большую выгоду.

Приступая к оптимизации, начинайте с самого верху. Для начала убедитесь, что используете наиболее оптимальный алгоритм, и лишь потом переходите к низкоуровневым трюкам при написании кода. Потенциальная выгода велика – возможно даже настолько, что вам, может быть, вообще не придется прибегать к каким-либо трюкам. Это утверждение нужно сочетать с предыдущим правилом: часто решение об алгоритме нужно принимать на ранней стадии написания программы.

13.2. Компиляция

Для управления процессом компиляции доступно пять параметров: `speed` (скорость производимого кода), `compilation-speed` (скорость самого

процесса компиляции), safety (количество проверок на ошибки в объектном коде), space (размер памяти, выделяемой для объектного кода) и debug (объем информации, оставляемой в коде для отладки).

Параметры компиляции не являются переменными в привычном понимании. Это своего рода веса, определяющие важность каждого параметра: от 0 (не важно) до 3 (очень важно). Представим, что узкое место находится во внутреннем цикле некоторой функции. Мы можем добавить декларацию следующего вида:

```
(defun bottleneck (...)
  (do (...)
      (...)
    (do (...)
        (...)
      (declare (optimize (speed 3) (safety 0)))
       ...)))
```

Как правило, подобные декларации добавляются лишь после завершения основной работы по созданию программы.

Чтобы глобально запросить наиболее быстрый компилируемый код, можно сказать:

```
(declaim (optimize (speed 3)
                   (compilation-speed 0)
                   (safety 0)
                   (debug 0)))
```

Это весьма радикальный и часто ненужный шаг. Не забывайте про бутылочное горлышко.[1]

Важным классом оптимизаций, производимых компиляторами Лиспа, является оптимизация хвостовых вызовов. Задавая максимальный вес параметра speed, вы включаете эту опцию, если она поддерживается компилятором.

Вызов считается *хвостовым*, если после него не нужно ничего вычислять. Следующая функция возвращает длину списка:

```
(defun length/r (lst)
  (if (null lst)
      0
      (1+ (length/r (cdr lst)))))
```

Данный рекурсивный вызов не является хвостовым, так как его значение передается функции 1+. А вот в следующей версии есть хвостовая рекурсия:

```
(defun length/tr (lst)
  (labels ((len (lst acc)
            (if (null lst)
```

[1] Старые реализации могут не предоставлять declaim. В этом случае используйте proclaim с цитируемым аргументом.

```
                acc
                (len (cdr lst) (1+ acc)))))
        (len lst 0)))
```

Вообще говоря, хвостовая рекурсия имеет здесь место в локальной функции `len`, поскольку рекурсивный вызов является последним действием в функции. Вместо того чтобы вычислять результирующее значение на обратном пути рекурсии, как это делает `length/r`, она аккумулирует его на пути вперед. Для этого и нужен аргумент `acc`, значение которого возвращается в конце рекурсивного вызова.

Хороший компилятор может преобразовывать хвостовой вызов в `goto`, и таким образом хвостовая рекурсия скомпилируется в обычный цикл.° В машинном коде, когда управление впервые передается той его части, которая реализует функцию `len`, на стек кладется информация, указывающая, что нужно сделать для завершения вызова. Но, поскольку после самого рекурсивного вызова делать ничего не надо, эти инструкции остаются действительными также и для второго вызова: для возврата из второго вызова нужно сделать то же самое, что и для возврата из первого. Таким образом, мы можем просто заменить старые аргументы функции новыми значениями, а после этого прыгнуть назад к началу функции и действовать так, как если бы это *был* второй вызов. При этом нет необходимости делать реальный вызов функции.

Другой способ уйти от затрат на полноценный вызов функции – заставить компилятор встраивать функции построчно (inline). Это особенно ценно для небольших функций, затраты на выполнение которых сопоставимы с затратами на сам вызов. Например, следующая функция выясняет, является ли ее аргумент списком или одиночным элементом:

```
(declaim (inline single?))

(defun single? (lst)
  (and (consp lst) (null (cdr lst))))
```

Интерактивный или интерпретируемый

Лисп – это интерактивный язык, но это не обязывает его быть интерпретируемым. Ранние версии Лиспа реализовывались как интерпретаторы, в результате чего сложилось мнение, что все преимущества Лиспа вытекают из его интерпретируемости. Эта идея ошибочна: Common Lisp является интерпретируемым и компилируемым языком одновременно.

Некоторые реализации Common Lisp и вовсе не используют интерпретацию. В них все, что вводится в toplevel, компилируется перед вычислением. Таким образом, называть toplevel интерпретатором не только старомодно, но и, строго говоря, ошибочно.

Так как до определения single? была сделана глобальная inline-декларация[1], использование single? не будет приводить к реальному вызову функции. Если мы определим вызывающую ее функцию так:

```
(defun foo (x)
  (single? (bar x)))
```

то при компиляции foo код single? будет встроен в код foo так, как если бы мы написали:

```
(defun foo (x)
  (let ((lst (bar x)))
    (and (consp lst) (null (cdr lst)))))
```

Существует два ограничения на inline-встраиваемость функции. Рекурсивные функции не могут быть встроены. И если inline-функция переопределяется, должны быть перекомпилированы все другие функции, вызывающие ее, иначе в них останется ее старое определение.

Для того чтобы избежать вызовов функций, в некоторых более ранних диалектах Лиспа использовались макросы (см. раздел 10.2). В Common Lisp делать это необязательно.

Различные компиляторы Лиспа отличаются друг от друга возможностями оптимизации. Чтобы узнать, какая работа реально проделана компилятором, полезно изучить скомпилированный код, который вы можете получить с помощью disassemble. Эта функция принимает функцию или имя функции и отображает результат ее компиляции, то есть набор машинных инструкций, которые реализуют эту функцию. Даже если дизассемблерный листинг является для вас китайской грамотой, вы можете хотя бы визуально оценить количество сделанных оптимизаций: скомпилируйте две версии – с оптимизирующими декларациями и без них – и просто оцените разницу. С помощью аналогичной методики можно выяснить, были ли функции встроены построчно. В любом случае, перед подобными экспериментами убедитесь, что установлены необходимые параметры компиляции для получения максимально быстрого кода.°

13.3. Декларации типов

Если Лисп – не первый язык программирования, с которым вы сталкиваетесь, то вас может удивить, что до сих пор мы ни разу не использовали то, что *совершенно необходимо* во многих других языках: декларации типов.

В большинстве языков для каждой переменной необходимо определить свой тип, и в дальнейшем переменная может содержать лишь значения этого типа. Такие языки называют языками с *сильной типизацией*.

[1] Чтобы inline-декларации были учтены, возможно, понадобится также установить параметры компиляции для генерации быстрого кода.

Такой подход заставляет программиста выполнять лишнюю работу и ограничивает его возможности: вы не можете писать функции, применимые к различным типам аргументов, а также хранить в структурах данных разные типы.° Однако преимуществом данного подхода является упрощение задачи компилятора: если он видит функцию, то знает заранее, какие действия нужно совершить.

В разделе 2.15 упоминалось, что Common Lisp использует более гибкий подход, называемый декларативной типизацией (manifest typing).[1] Типы имеют значения, а не переменные. Последние могут содержать объекты любых типов.

Если бы мы на этом остановились, то вынуждены были бы платить скоростью за гибкость. Поскольку функция + может принимать аргументы разных типов, при каждом выполнении она должна затрачивать дополнительное время на выяснение того, с данными каких типов она вызывается.

Если, например, от функции требуется всего лишь сложение целых чисел, отказ от ее оптимизации приведет к низкой производительности. Подход к решению данной проблемы в Common Lisp таков: сообщите все, что вам известно. Если вам заранее известно, что вы складываете два числа типа `fixnum`, то можно объявить их таковыми, и компилятор сгенерирует код целочисленного сложения такой же, как в C.

Таким образом, различие в подходе к оптимизации не приводит к разнице в плане скорости. Просто первый подход требует всех деклараций типов, а второй – нет. В Common Lisp объявления типов совершенно не обязательны. Они могут ускорить работу программы, но (если, конечно, они сами корректны) не способны изменить ее поведение.

Глобальные декларации выполняются с помощью `declaim`, за которой должна следовать хотя бы одна декларационная форма. Декларация типа – это список, содержащий тип символа, сопровождаемый именем типа и именами одной или более переменных. Таким образом, для объявления типа глобальной переменной достаточно сказать:

```
(declaim (type fixnum *count*))
```

ANSI Common Lisp допускает декларации без использования слова `type`:

```
(declaim (fixnum *count*))
```

Локальные декларации выполняются с помощью `declare`, которая принимает те же аргументы, что и `declaim`. Декларации могут начинать любое тело кода, в котором появляются новые переменные: `defun`, `lambda`,

---

[1] Применяемый в Лиспе подход к типизации можно описать двумя способами: по месту хранения информации о типах и по месту ее применения. Декларативная типизация подразумевает связывание информации о типе с объектом данных, а *типизация времени выполнения* (*run-time typing*) подразумевает, что информация о типах используется лишь в процессе выполнения программы. По сути, это одно и то же.

let, do и другие. К примеру, чтобы сообщить компилятору, что параметры функции принадлежат типу fixnum, нужно сказать:

```
(defun poly (a b x)
  (declare (fixnum a b x))
  (+ (* a (expt x 2)) (* b x)))
```

Имя переменной в декларации ссылается на переменную, действительную в том же контексте, где встречается сама декларация.

Вы можете также задать тип любого выражения в коде с помощью the. Например, если нам известно, что значения a, b и x не только принадлежат типу fixnum, но и достаточно малы, чтобы промежуточные выражения также принадлежали типу fixnum, вы можете указать это явно:

```
(defun poly (a b x)
  (declare (fixnum a b x))
  (the fixnum (+ (the fixnum (* a (the fixnum (expt x 2))))
                 (the fixnum (* b x)))))
```

Выглядит довольно неуклюже, не так ли? К счастью, есть две причины, по которым вам редко понадобится шпиговать численный код объявлениями the. Во-первых, это лучше поручить макросам.° Во-вторых, многие реализации используют особые трюки, чтобы ускорить целочисленную арифметику независимо от деклараций.

В Common Lisp существует невероятное многообразие типов; их набор практически не ограничен, ведь вы можете самостоятельно объявлять собственные типы. Однако явно объявлять типы имеет смысл только в некоторых случаях. Вот два основных правила, когда это стоит делать:

1. Имеет смысл декларировать типы в тех функциях, которые могут работать с аргументами некоторых разных типов (но не всех). Если вам известно, что аргументы вызова функции + всегда будут fixnum или первый аргумент aref всегда будет массивом одного типа, декларация будет полезной.
2. Обычно имеет смысл декларировать лишь те типы, которые находятся внизу иерархии типов: объявления с fixnum или simple-array будут полезными, а вот декларации integer или sequence не принесут ощутимого результата.

Декларации типов особенно важны при работе со сложными объектами, включая массивы, структуры и экземпляры. Такие декларации не только приводят к более быстрому коду, но и позволяют более эффективно организовать объекты в памяти.

Если о типе элементов массива ничего неизвестно, то он представляется в памяти как набор указателей. Однако если тип известен и все элементы принадлежат к одному типу, скажем double-float, тогда массив может быть представлен как набор чисел в формате double-float. Во-первых, такой массив будет более экономно использовать память. Во-вторых, отсутствие необходимости переходить по указателям приведет к более быстрому чтению и записи элементов.

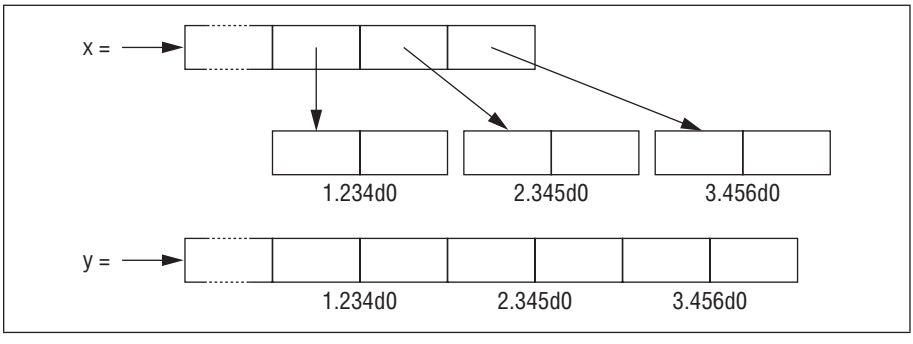

Рис. 13.1. *Результат задания типа элементов массива*

Тип массива можно задать с помощью аргумента `:element-type` в `make-array`. Такой массив называется *специализированным*. На рис. 13.1 показано, что будет происходить в большинстве реализаций при выполнении следующего кода:

```
(setf x (vector 1.234d0 2.345d0 3.456d0)
      y (make-array 3 :element-type 'double-float)
      (aref y 0) 1.234d0
      (aref y 1) 2.345d0
      (aref y 2) 3.456d0)
```

Каждый прямоугольник на рисунке соответствует машинному слову в памяти. Каждый из двух массивов содержит заголовок неопределенной длины, за которым следует представление трех элементов. В массиве `x` каждый элемент – указатель. В нашем случае все три указателя одновременно ссылаются на элементы `double-float`, но могут ссылаться на произвольные объекты. В массиве `y` элементы действительно являются числами `double-float`. Второй вариант работает быстрее и занимает меньше места, но мы вынуждены платить за это ограничением на однородность массива.

Заметьте, что для доступа к элементам `y` мы пользовались `aref`. Специализированный вектор больше не принадлежит типу `simple-vector`, поэтому мы не можем ссылаться на его элементы с помощью `svref`.

При создании массива в коде, который его использует, необходимо помимо специализации объявить размерность и тип элемента. Такая декларация будет выглядеть следующим образом:

```
(declare (type (vector fixnum 20) v))
```

Эта запись говорит о том, что вектор `v` имеет размерность 20 и специализирован для целых чисел типа `fixnum`.

Наиболее общая декларация включает тип массива, тип элементов и список размерностей:

```
(declare (type (simple-array fixnum (4 4)) ar))
```

Массив ar теперь считается простым массивом 4×4, специализированным для fixnum.

На рис. 13.2 показано, как создать массив 1000×1000 элементов типа single-float и как написать функцию, суммирующую все его элементы. Массивы располагаются в памяти в построчном порядке (row-major order). Рекомендуется по возможности проходить по элементам массивов в таком же порядке.

```
(setf a (make-array '(1000 1000)
                    :element-type 'single-float
                    :initial-element 1.0s0))

(defun sum-elts (a)
  (declare (type (simple-array single-float (1000 1000))
                 a))
    (let ((sum 0.0s0))
      (declare (type single-float sum))
      (dotimes (r 1000)
        (dotimes (c 1000)
          (incf sum (aref a r c))))
   sum))
```

Рис. 13.2. *Суммирование по массиву*

Чтобы сравнить производительность sum-elts с декларациями и без них, воспользуемся макросом time. Он измеряет время, необходимое для вычисления выражения, причем в разных реализациях результаты его работы отличаются. Его применение имеет смысл только для скомпилированных функций. Если мы скомпилируем sum-elts с параметрами, обеспечивающими максимально быстрый код, time вернет менее полсекунды:

```
> (time (sum-elts a))
User Run Time = 0.43 seconds
1000000.0
```

Если же мы теперь уберем все декларации и перекомпилируем sum-elts, то те же вычисления займут больше пяти секунд:

```
> (time (sun-elts a))
User Run Time = 5.17 seconds
1000000.0
```

Важность деклараций типов, особенно при работе с массивами и отдельными числами, сложно переоценить. В данном случае две строчки кода обеспечили нам двенадцатикратный прирост производительности.

13.4. Обходимся без мусора

Лисп позволяет отложить решения касательно не только типов переменных, но и выделения памяти. На ранних стадиях создания программы возможность не думать о таких вещах дает больше свободы воображению. Но по мере того как программа становится зрелой, она может стать быстрее, меньше полагаясь на динамическое выделение памяти.

Однако уменьшение консинга (consing) не обязательно приведет к ускорению программы. В реализациях Лиспа с плохими сборщиками мусора такие программы обычно работают медленно. До недавнего времени большинство реализаций имели не самые удачные сборщики мусора, поэтому бытовало мнение, что эффективные программы должны по возможности избегать выделения памяти. Но последние усовершенствования перевернули ситуацию с ног на голову. Некоторые реализации теперь имеют настолько изощренные сборщики мусора, что выделять память под объекты и выбрасывать их за ненадобностью оказывается быстрее, чем использовать эту память повторно.

В этом разделе показаны основные методы экономии памяти. Скажется ли это на производительности, зависит от используемой реализации. Как всегда, лучший совет: попробуйте сами – и все увидите.

Существует множество приемов, позволяющих избежать выделения памяти. Некоторые из них практически не изменяют вид вашей программы. Например, одним из наиболее простых методов является использование деструктивных функций. В следующей таблице приводятся некоторые часто употребляемые функции и их деструктивные аналоги:

Безопасные	**Деструктивные**
`append`	`nconc`
`reverse`	`nreverse`
`remove`	`delete`
`remove-if`	`delete-if`
`remove-duplicates`	`delete-duplicates`
`subst`	`nsubst`
`subst-if`	`nsubst-if`
`union`	`nunion`
`intersection`	`nintersection`
`set-difference`	`nset-difference`

Если вам известно, что модификация списка безопасна, используйте `delete` вместо `remove`, `nreverse` вместо `reverse` и т. д.

Если вы желаете полностью исключить выделение памяти, это еще не значит, что придется забыть о возможности создания объектов на лету. Чего следует избегать, так это выделения памяти под объекты на лету

и ее последующего сбора. Общим решением является заблаговременное выделение блоков памяти и их дальнейшее повторное использование «вручную». *Заблаговременно* означает в момент компиляции или некоторой процедуры инициализации. Когда скорость начинает играть роль, зависит от приложения.

Например, если обстоятельства позволяют нам ограничить размер стопки, то вместо ее создания из отдельных ячеек можно сделать так, что стопка будет расти и уменьшаться в рамках заранее выделенного вектора. Common Lisp имеет встроенную поддержку использования вектора в качестве стопки. Для этого есть необязательный параметр `fill-pointer` в `make-array`. Первый аргумент `make-array` задает количество памяти, выделяемой под вектор, а `fill-pointer`, если задан, определяет его исходную фактическую длину:

```
> (setf *print-array* t)
T
> (setf vec (make-array 10 :fill-pointer 2
                           :initial-element nil))
#(NIL NIL)
```

Функции работы с последовательностями будут рассматривать его как вектор из двух элементов:

```
> (length vec)
2
```

но его размер может вырасти до 10. Поскольку этот вектор имеет указатель заполнения, к нему можно применять функции `vector-push` и `vector-pop`, аналогичные `push` и `pop` для списков:

```
> (vector-push 'a vec)
2
> vec
#(NIL NIL A)
> (vector-pop vec)
A
> vec
#(NIL NIL)
```

Вызов `vector-push` увеличил указатель заполнения и вернул его старое значение. Мы можем записывать в такой вектор новые значения до тех пор, пока указатель заполнения меньше начального размера вектора, заданного первым аргументом `make-array`. Когда свободное место закончится, `vector-push` вернет `nil`. В наш вектор `vec` мы можем поместить до 8 новых элементов.

Векторы с указателем заполнения имеют один недостаток – они более не принадлежат типу `simple-vector`, и вместо `svref` нам приходится использовать `aref`. Эти дополнительные затраты стоит учесть при выборе подходящего решения.

Некоторые приложения могут производить очень длинные последовательности. В таком случае вам поможет использование map-into вместо map. В качестве первого аргумента она получает не тип новой последовательности, а саму последовательность, в которую будут записаны результаты. Из этой же последовательности могут браться и аргументы для функции, которая производит отображение. Для примера увеличим на единицу все элементы вектора:

```
(setf v (map-into v #'1+ v))
```

На рис. 13.3 показан пример приложения, работающего с длинными векторами. Это программа, генерирующая несложный словарь рифм (точнее, словарь ранее увиденных рифм).

```
(defconstant dict (make-array 25000 :fill-pointer 0))

(defun read-words (from)
  (setf (fill-pointer dict) 0)
  (with-open-file (in from :direction :input)
    (do ((w (read-line in nil :eof)
            (read-line in nil :eof)))
        ((eql w :eof))
      (vector-push w dict))))

(defun xform (fn seq) (map-into seq fn seq))

(defun write-words (to)
  (with-open-file (out to :direction :output
                          :if-exists :supersede)
    (map nil #'(lambda (x)
                 (fresh-line out)
                 (princ x out))
             (xform #'nreverse
                    (sort (xform #'nreverse dict)
                          #'string<)))))
```

Рис. 13.3. *Генерация словаря рифм*

Функция read-words читает слова из файла, содержащего по слову на строке,° а write-words печатает их в обратном алфавитном порядке. Ее вывод может начинаться так:

```
a amoeba alba samba marimba ...
```

и завершаться так:

```
... megahertz gigahertz jazz buzz fuzz
```

Используя преимущества векторов с указателем заполнения и map-into, мы сможем легко и эффективно написать эту программу.

В расчетных приложениях будьте аккуратны с типами bignum. Операции с bignum требуют выделения памяти и работают существенно медленнее. Но даже если ваша программа должна в конце возвращать значения bignum, ее можно сделать эффективнее, организовав промежуточные результаты как значения fixnum.

Другой способ избежать выделения памяти – заставить компилятор размещать объекты не в куче, а на стеке. Если вам известно, что какой-либо объект будет использоваться временно, вы можете избежать выделения для него памяти в куче с помощью декларации *dynamic extent*.

Декларируя dynamic-extent для переменной, вы утверждаете, что ее значение не будет жить дольше, чем сама переменная. Как может значение существовать дольше переменной? Вот пример:

```
(defun our-reverse (lst)
  (let ((rev nil))
    (dolist (x lst)
      (push x rev))
    rev))
```

Функция our-reverse аккумулирует переданный ей список в обратном порядке в rev и по завершении возвращает ее значение. Сама переменная исчезает, а список, который в ней находился, продолжает существовать и возвращается назад вызвавшей функции. Дальнейшая его судьба непредсказуема.

Для контраста рассмотрим следующую реализацию adjoin:

```
(defun our-adjoin (obj lst &rest args)
  (if (apply #'member obj lst args)
      lst
      (cons obj lst)))
```

Из этого определения функции видно, что список args никуда не передается. Он нужен не дольше, чем сама переменная. Значит, в этой ситуации имеет смысл сделать декларацию dynamic-extent:

```
(defun our-adjoin (obj lst &rest args)
  (declare (dynamic-extent args))
  (if (apply #'member obj lst args)
      lst
      (cons obj lst)))
```

Теперь компилятор может (но не обязан) размещать args на стеке, то есть это значение будет автоматически отброшено по выходу из our-adjoin.

13.5. Пример: заранее выделенные наборы

Чтобы избежать динамического выделения памяти в приложении, которое использует структуры данных, вы можете заранее разместить конкретное их количество в *пуле* (*pool*). Когда вам необходима структура,

вы берете ее из пула, а по завершении работы с ней возвращаете структуру назад в пул.° Чтобы проиллюстрировать эту идею, напишем прототип программы, контролирующей перемещение кораблей в гавани, а затем перепишем этот прототип с использованием пула.

На рис. 13.4 показана первая версия. Глобальная переменная `*harbour*` содержит список кораблей, каждый из которых представлен в виде структуры `ship`. Функция `enter` вызывается при заходе корабля в порт; `find-ship` находит корабль с заданным именем (если он существует); `leave` вызывается, когда корабль покидает гавань.

```
(defparameter *harbor* nil)

(defstruct ship
  name flag tons)

(defun enter (n f d)
  (push (make-ship :name n :flag f :tons d)
        *harbor*))

(defun find-ship (n)
  (find n *harbor* :key #'ship-name))

(defun leave (n)
  (setf *harbor*
        (delete (find-ship n) *harbor*)))
```

Рис. 13.4. *Порт*

Отличный подход для написания первой версии программы, но такая реализация производит достаточно много мусора. При выполнении программы память выделяется двумя путями: при заходе кораблей в гавань будут производиться новые структуры, а с ростом списка `*harbour*` будут выделяться новые ячейки.

Мы можем избавиться от обоих источников мусора, выделяя память в момент компиляции. На рис. 13.5 приведена вторая версия программы, которая не производит никакого мусора.

Строго говоря, выделение памяти все же происходит, но не в момент выполнения. Во второй версии `*harbour*` является хеш-таблицей, а не списком, и пространство под нее выделяется при компиляции. Тысяча структур `ship` также создается во время компиляции и сохраняется в векторе `pool`. (Если параметр `:fill-pointer` имеет значение `t`, указатель заполнения размещается в конце вектора.) Теперь при вызове `enter` нам не нужно создавать новую структуру, достаточно получить одну из уже существующих в пуле с помощью `make-ship`. А когда `leave` удаляет корабль из гавани, соответствующая структура не становится мусором, а возвращается назад в пул.

```
(defconstant pool (make-array 1000 :fill-pointer t))

(dotimes (i 1000)
  (setf (aref pool i) (make-ship)))

(defconstant harbor (make-hash-table :size 1100
                                     :test #'eq))

(defun enter (n f d)
  (let ((s (if (plusp (length pool))
               (vector-pop pool)
               (make-ship))))
    (setf (ship-name s)        n
          (ship-flag s)        f
          (ship-tons s)        d
          (gethash n harbor) s)))

(defun find-ship (n) (gethash n harbor))

(defun leave (n)
  (let ((s (gethash n harbor)))
    (remhash n harbor)
    (vector-push s pool)))
```

Рис. 13.5. *Порт, версия 2*

Используя пулы, мы собственноручно выполняем часть работы по управлению памятью. Сделает ли это нашу программу быстрее, зависит от того, как конкретная реализация управляет памятью. Грубо говоря, использование пулов оправдано лишь в реализациях с примитивными сборщиками мусора, а также в приложениях реального времени, где непредсказуемый запуск сборщика мусора может вызвать проблемы.

13.6. Быстрые операторы

В начале главы было сказано, что Лисп – это, по сути, два разных языка. Если вы приглядитесь к дизайну Common Lisp, то увидите, что часть его операторов предназначена для ускорения выполнения, а другая часть – для удобства разработки.

Например, для доступа к элементу вектора существуют три оператора: `elt`, `aref`, `svref`. Такое разнообразие позволяет выжать из программы максимум производительности. На тех участках, где важна скорость, используйте `svref` вместо `elt`, которая работает и с массивами, и со списками.

Для работы со списками эффективнее использовать специализированную функцию `nth`, нежели `elt`. Лишь одна функция, `length`, не имеет аналогов для разных типов. Почему в Common Lisp нет отдельной версии этой функции для списков? Потому что программа, выполняющая

подсчет длины списка, уже безнадежна в плане производительности. В этом случае, как и во многих других, сам дизайн языка объясняет, что является эффективным, а что – нет.

Другая пара похожих функций – `eql` и `eq`. Первый предикат проверяет на идентичность, второй – на одинаковое размещение в памяти. Второй предикат обеспечивает большую эффективность, однако его стоит использовать, если только вам заранее известно, что аргументы не являются числами или знаками. Два объекта равны с точки зрения `eq`, если они имеют одинаковое размещение в памяти. Числа и знаки не обязаны располагаться в каком-либо определенном месте в памяти, поэтому `eq` к ним неприменима (хотя во многих реализациях `eq` работает с типом `fixnum`). Для любых других аргументов `eq` будет работать аналогично `eql`.

Разумеется, быстрее всего выполнять сравнение объектов с помощью `eq`, так как при этом Лиспу достаточно сравнить лишь два указателя. Это значит, что хеш-таблицы с тестовой функцией `eq` (см. рис. 13.5) обеспечивают самый быстрый доступ. В таких таблицах `gethash` хеширует лишь указатели и даже не смотрит, на что они указывают. Помимо скорости доступа с хеш-таблицами связан еще один момент. Использование `eq`- и `eql`-таблиц приводит к дополнительным издержкам в случае применения копирующей сборки мусора, поскольку после каждой сборки хеши таких таблиц должны быть пересчитаны. Если это вызывает проблемы, лучше использовать `eql`-таблицы и числа типа `fixnum` в качестве ключей.

Вызов `reduce` может быть эффективнее вызова `apply`,[1] когда функция принимает остаточный параметр. Например, вместо выражения

```
(apply #'+ '(1 2 3))
```

эффективнее использовать следующее выражение:

```
(reduce #'+ '(1 2 3))
```

Кроме того, важно не только вызывать правильные функции, но и вызывать их правильно. Необязательные аргументы, аргументы по ключу и остаточные аргументы дорогостоящи. В случае обычных аргументов вызывающая сторона кладет их в заранее известное вызванной функции место, в то время как для других аргументов требуется дополнительная обработка на этапе выполнения. Хуже всего в этом отношении дело обстоит с аргументами по ключу. Для встроенных функций, использующих эти аргументы, хорошие компиляторы используют особые подходы, чтобы получить быстрый код[2]. Но в ваших собственных функциях рекомендуется избегать использования аргументов по ключу на

---

[1] Это утверждение автора является довольно сомнительным. По крайней мере, для ряда современных реализаций на приведенном примере наблюдается выраженный противоположный эффект. – *Прим. перев.*

[2] Речь идет о таких техниках, как, например, прямое кодирование (open coding). – *Прим. перев.*

критических в плане скорости участках программы. Также разумно не передавать много значений через остаточный аргумент там, где этого можно избежать.

Индивидуальные компиляторы иногда применяют особые оптимизации. Например, некоторые могут оптимизировать вызов `case` с целочисленными ключами в небольшом диапазоне их значений. Как правило, узнать подробнее о специфических оптимизациях в вашей реализации вы можете из документации к ней.

13.7. Две фазы разработки

Если в вашем приложении производительность имеет первостепенное значение, в качестве одного из решений попробуйте переписать критические участки на низкоуровневом языке, например на C или ассемблере. Такой подход применим не только к Лиспу, но и к любому высокоуровневому языку – критические участки программы на C часто переписываются на ассемблере, – но чем абстрактнее язык, тем больше преимуществ дает разбиение разработки на две фазы.

Стандарт Common Lisp не описывает механизм интеграции кода, написанного на других языках. Эта задача полностью ложится на плечи разработчиков конкретной реализации, и почти все из них так или иначе решают ее.[1]

Вам может показаться напрасной тратой времени написание части кода на одном языке с последующим переписыванием ее на другом. Тем не менее практика показала, что это отличный способ написания приложений. Оказалось, что проще сначала разработать функциональность программы, а затем ее оптимизировать, чем делать это одновременно.

Если бы программирование было всего лишь механическим процессом, трансляцией спецификаций в код, было бы разумно выполнять всю работу за один шаг. Но настоящее программирование – это совсем иное. Независимо от проработанности спецификаций исследовательский компонент в написании программ очень важен, и часто намного важнее, чем этого ожидают.

Казалось бы, если спецификации были *хорошими*, то программирование заключалось *бы* лишь в кодировании. Это порочная идея, хотя и распространенная. Исследовательская компонента важна уже хотя бы потому, что любая спецификация по определению содержит недостатки. Иначе она не была бы спецификацией.

В других областях точность спецификации может иметь первостепенное значение. Если вы просите выточить из куска металла определен-

[1] Библиотеки CFFI и UFFI предоставляют обобщенный механизм использования интерфейса с внешним кодом (foreign function interface), доступный в разных реализациях. – *Прим. перев.*

ную форму, необходимо как можно точнее описать, что вы хотите получить. Но это правило не распространяется на программы, поскольку и спецификации, и программный код делаются из одного и того же материала: текста. Вы *не можете* разработать спецификацию, которая в точности описывает все требования задачи. В противном случае это была бы уже готовая программа.°

В приложениях, подразумевающих значительное количество исследований (объем которых опять же превзойдет ожидания), имеет смысл разделять реализацию программы на две фазы. Промежуточный результат после первой фазы не будет окончательным. Например, при ваянии бронзовой скульптуры принято делать первый набросок из глины, который затем используется для создания формы, в которую будет отлита скульптура из бронзы.° И хотя в завершенной скульптуре глины не остается, эффект от ее использования все равно заметен. Теперь вообразите, насколько сложней была бы та же задача при наличии лишь слитка бронзы и зубила. По тем же причинам удобнее написать программу на Лиспе, а затем переписать на С, чем писать ее на С с самого начала.

Итоги главы

1. К оптимизации не следует приступать раньше времени. Основное внимание нужно уделять узким местам и начинать следует с выбора оптимального алгоритма.
2. Доступно пять параметров управления компиляцией. Они могут устанавливаться как глобальными, так и локальными декларациями.
3. Хорошие компиляторы могут оптимизировать хвостовую рекурсию, превращая ее в циклы. Встраивание кода позволяет избежать вызова функций.
4. Декларации типов не обязательны, но могут сделать программу более производительной. Особенно важны декларации в расчетном коде и операциях с массивами.
5. Выделение меньшего количества памяти позволяет ускорить программу, особенно в реализациях с примитивными сборщиками мусора. Для этого используйте деструктивные функции, предварительное выделение памяти и размещение объектов на стеке.
6. В ряде ситуаций полезно брать объекты из предварительно созданного пула.
7. Некоторые части Лиспа предназначены для быстрой работы, некоторые – для гибкой.
8. Процесс программирования обязательно включает исследовательский момент, который следует отделять от оптимизации, иногда даже вплоть до использования двух разных языков.

Упражнения

1. Проверьте, применяет ли ваш компилятор `inline`-декларации.
2. Перепишите следующую функцию с использованием хвостовой рекурсии. Скомпилируйте и сравните производительность обеих версий.

```
(defun foo (x)
  (if (zerop x)
      0
      (+ 1 (foo (1- x)))))
```

   Подсказка: вам потребуется еще один аргумент.
3. Добавьте декларации в следующие программы. Насколько ускорится их выполнение?
   (a) Арифметика дат в разделе 5.7.
   (b) Трассировщик лучей в разделе 9.8.
4. Перепишите код поиска в ширину в разделе 3.15 так, чтобы он выделял как можно меньше памяти.
5. Переделайте код двоичных деревьев поиска (см. раздел 4.7) с использованием пулов.

14

Более сложные вопросы

Эта глава необязательна. В ней описывается ряд эзотерических особенностей языка. Common Lisp похож на айсберг: огромная часть его возможностей не видна для большинства пользователей. Быть может, вам никогда не придется определять пакеты или макросы чтения самостоятельно, но знание подобных моментов может сильно облегчить жизнь.

14.1. Спецификаторы типов

В Common Lisp типы не являются объектами. Так, к примеру, нет объекта, соответствующего типу `integer`. То, что мы получаем при вызове `type-of` и передаем в качестве аргумента функции `typep`, является не типом, а спецификатором типа.

Спецификатор типа – это его имя. Простейшими спецификаторами типов являются символы типа `integer`. Они формируют иерархию типов во главе с типом `t`, к которому принадлежат все объекты. Иерархия типов не является деревом. Например, от `nil` наверх ведут два пути: один через `atom`, а второй через `list` и `sequence`.

На самом деле, тип – это просто множество объектов. Это означает, что количество типов, как и количество множеств объектов, может быть бесконечным. Некоторые типы обозначаются атомарно, например `integer` определяет множество целых чисел. Но мы можем также конструировать составные спецификаторы типов, которые ссылаются на любые множества объектов.

Например, пусть *a* и *b* – спецификаторы некоторых типов, тогда `(or` *a b*`)` обозначает объединение множеств объектов, соответствующих типам *a* и *b*. Таким образом, объект будет принадлежать типу `(or` *a b*`)`, если он принадлежит типу *a* или типу *b*.

Если бы circular? была функцией, возвращающей истину для циклических по хвосту списков, то для определения набора правильных последовательностей можно было бы использовать следующий спецификатор:[1]

```
(or vector (and list (not (satisfies circular?))))
```

Некоторые атомарные спецификаторы типов могут встречаться и в составных типах. Например, следующий спецификатор соответствует набору целых чисел от 1 до 100 включительно:

```
(integer 1 100)
```

В таких случаях говорят, что спецификатор определяет *конечный тип*.

В составном спецификаторе типа некоторые поля могут оставаться неопределенными. Такое поле помечается * вместо параметра. Так,

```
(simple-array fixnum (* *))
```

описывает набор двумерных простых массивов, специализированных для fixnum, а

```
(simple-array fixnum *)
```

описывает множество (надтип предыдущего) простых массивов, специализированных для fixnum. Завершающие звездочки могут быть опущены, поэтому последний спецификатор можно записать так:

```
(simple-array fixnum)
```

Если составной тип не содержит аргументов, то он эквивалентен атомарному. Таким образом, тип simple-array описывает множество всех простых массивов.

Чтобы постоянно не повторять громоздкие определения составных спецификаторов, которые вы часто используете, для них можно определить аббревиатуры с помощью deftype. Его использование напоминает defmacro, только он раскрывается не в выражение, а в спецификатор типа. С помощью следующего определения:

```
(deftype proseq ()
  '(or vector (and list (not (satisfies circular?)))))
```

мы создаем новый атомарный спецификатор proseq:

```
> (typep #(1 2) 'proseq)
T
```

Если вы определите составной спецификатор с аргументами, то они не будут вычисляться, как и в случае defmacro. Так,

```
(deftype multiple-of (n)
  `(and integer (satisfies (lambda (x)
                             (zerop (mod x ,n))))))
```

[1] Несмотря на то, что этот факт обычно не упоминается в стандарте, вы можете использовать в спецификаторах типов выражения and и or с любым количеством аргументов, подобно макросам and и or.

определяет (multiple-of *n*) как спецификатор для всех множителей *n*:

```
> (typep 12 '(multiple-of 4))
T
```

Спецификаторы типов интерпретируются и поэтому работают медленно, так что в общем случае для подобных проверок лучше определить функцию.

14.2. Бинарные потоки

В главе 7 помимо потоков знаков упоминались также и бинарные потоки. Бинарный поток – это источник и/или получатель не знаков, а *целых чисел*. Для создания бинарного потока нужно задать подтип integer (чаще всего это unsigned-byte) как значение параметра :element-type при открытии потока.

Для работы с бинарными потоками имеются лишь две функции: read-byte и write-byte. Функцию для копирования содержимого файла можно определить следующим образом:

```
(defun copy-file (from to)
  (with-open-file (in from :direction :input
                           :element-type 'unsigned-byte)
    (with-open-file (out to :direction :output
                            :element-type 'unsigned-byte)
      (do ((i (read-byte in nil -1)
              (read-byte in nil -1)))
          ((minusp i))
        (declare (fixnum i))
        (write-byte i out)))))
```

Задавая тип unsigned-byte в качестве аргумента :element-type, вы указываете операционной системе тип элементов, для которых будет производиться ввод-вывод. При желании работать конкретно, к примеру, с 7-битными числами, можно передать в :element-type

```
(unsigned-byte 7)
```

14.3. Макросы чтения

В разделе 7.5 была представлена концепция макрознаков – знаков, имеющих особое значение для read. С каждым из них связана функция, указывающая read, что делать при встрече этого знака. Вы можете изменять определения функций, связанных с уже существующими макрознаками, а также определять собственные макросы чтения.

Функция set-macro-character предоставляет один из способов определения макросов чтения. Она принимает знак и функцию, вызываемую read при встрече этого знака.

Одним из старейших макросов чтения в Лиспе является ', quote. Мы могли бы определить его следующим образом:

```
(set-macro-character #\'
  #'(lambda (stream char)
      (list (quote quote) (read stream t nil t))))
```

Когда read встречает ' в обычном контексте, она возвращает результат вызова данной функции для текущего состояния потока и знака. (В данном случае функция игнорирует свой второй аргумент, так как он всегда является кавычкой.) Поэтому когда read увидит 'a, она вернет (quote a).

Теперь можно понять смысл последнего аргумента read. Он сообщает, произведен ли вызов read внутри другого вызова read. Следующие аргументы read будут одинаковыми практически для всех макросов чтения: первый аргумент – поток; второй аргумент, t, сообщает, что при достижении конца файла read будет сигнализировать об ошибке; третий аргумент задает значение, возвращаемое вместо данной ошибки в случае истинности второго аргумента (в нашем примере он не применяется); четвертый аргумент, t, сообщает о рекурсивности вызова read.

Вы можете определять (с помощью make-dispatch-macro-character) свои собственные диспетчеризирующие макрознаки, но раз # уже определен для этой цели, можно пользоваться и им. Шесть комбинаций, начинающихся с #, явным образом зарезервированы для ваших нужд: #!, #?, #[, #], #{ и #}.

Новую комбинацию с управляющим макрознаком можно определить с помощью вызова функции set-dispatch-macro-character, которая схожа с set-macro-character, но set-dispatch-macro-character принимает не один, а два знака. Следующий код определяет #? в качестве макроса чтения, возвращающего список целых чисел:

```
(setf-dispatch-macro-character #\# #\?
  #'(lambda (stream char1 char2)
      (list 'quote
            (let ((lst nil))
              (dotimes (i (+ (read stream t nil t) 1))
                (push i lst))
              (nreverse lst)))))
```

Теперь #?*n* будет прочитан как список всех целых чисел от 0 до *n*. Например:

```
> #?7
(0 1 2 3 4 5 6 7)
```

Второе место по распространенности после простых макрознаков занимают знаки-ограничители списков. Еще одна комбинация, зарезервированная для пользователя, – #{. Вот пример определения более продвинутых скобок:

```
(set-macro-character #\} (get-macro-character #\)))
```

```
(set-dispatch-macro-character #\# #\{
  #'(lambda (stream char1 char2)
      (let ((accum nil)
            (pair (read-delimited-list #\} stream t)))
        (do ((i (car pair) (+ i 1)))
            ((> i (cadr pair))
             (list 'quote (nreverse accum)))
          (push i accum)))))
```

Они задают чтение выражения #{x y} в качестве списка целых чисел от x до y включительно:

```
> #{2 7}
(2 3 4 5 6 7)
```

Функция read-delimited-list определена специально для таких макросов чтения. Ее первым аргументом является знак, который расценивается как конец списка. Чтобы знак } был опознан как разграничитель, необходимо предварительно сообщить об этом с помощью set-macro-character.

Чтобы иметь возможность пользоваться макросом чтения в файле, в котором определен сам макрос, его определение должно быть завернуто в выражение eval-when с целью обеспечить его выполнение в момент компиляции. В противном случае определение будет скомпилировано, но не вычислено до тех пор, пока скомпилированный файл не будет загружен.

14.4. Пакеты

Пакет – это объект Лиспа, сопоставляющий именам символы. Текущий пакет всегда хранится в глобальной переменной *package*. При запуске Common Lisp стартовым пакетом является common-lisp-user, неформально известный также как пользовательский пакет. Функция package-name возвращает имя текущего пакета, а find-package возвращает пакет с заданным именем:

```
> (package-name *package*)
"COMMON-LISP-USER"
> (find-package "COMMON-LISP-USER")
#<Package "COMMON-LISP-USER" 4CD15E>
```

Обычно символ интернируется в пакет, являющийся текущим на момент его чтения. Функция symbol-package принимает символ и возвращает пакет, в который он был интернирован.

```
> (symbol-package 'sym)
#<Package "COMMON-LISP-USER" 4CD15E>
```

Интересно, что это выражение вернуло именно такое значение, поскольку оно должно было быть считано перед вычислением, а само считывание привело к интернированию sym. Для последующего применения присвоим sym некоторое значение:

```
> (setf sym 99)
99
```

Теперь мы создадим новый пакет и переключимся в него:

```
> (setf *package* (make-package 'mine
                                :use '(common-lisp)))
#<Package "MINE" 63390E>
```

В этот момент должна зазвучать зловещая музыка, ибо теперь мы в ином мире, где sym перестал быть тем, чем был раньше:

```
MINE> sym
Error: SYM has no value.
```

Почему это произошло? Чуть раньше мы установили sym в 99, но сделали это в другом пакете, а значит, для другого символа, нежели sym в пакете mine.[1] Чтобы сослаться на исходный sym из другого пакета, необходимо его предварить именем его родного пакета и парой двоеточий:

```
MINE> common-lisp-user::sym
99
```

Итак, несколько символов с одинаковыми именами могут сосуществовать, если находятся в разных пакетах. Один символ находится в пакете common-lisp-user, другой – в mine, и это разные символы. В этом заключается весь смысл пакетов. Поместив свою программу в отдельный пакет, вы можете не переживать, что кто-то использует выбранные вами имена функций и переменных где-то еще. Даже если где-либо будет использовано такое же имя, как у вас, это будет уже другой символ.

Пакеты также предоставляют возможность сокрытия информации. Программы должны ссылаться на функции и переменные по их именам. Если вы не делаете какое-либо имя доступным вне своего пакета, то вряд ли функции из других пакетов смогут использовать и изменять то, на что это имя ссылается.

Использование пары двоеточий для обозначения символа не из текущего пакета считается плохим стилем. Фактически вы нарушаете модульность, которая устанавливается пакетами. Если вам приходится использовать двойное двоеточие для ссылки на символ, это означает, что кто-то не хочет, чтобы вы на него ссылались.

Обычно следует ссылаться лишь на *экспортированные* символы. Если мы вернемся назад в пользовательский пакет (in-package устанавливает *package*) и экспортируем символ, интернированный в нем,

```
MINE> (in-package common-lisp-user)
#<Package "COMMON-LISP-USER" 4CD15E>
> (export 'bar)
T
```

[1] Некоторые реализации Common Lisp печатают имя пакета перед приглашением toplevel, когда вы находитесь не в пользовательском пакете.

```
> (setf bar 5)
5
```

то сделаем его видимым и в других пакетах. Теперь, вернувшись в пакет mine, мы сможем ссылаться на bar через одинарное двоеточие, поскольку отныне это публично доступное имя:

```
> (in-package mine)
#<Package "MINE" 63390E>
MINE> common-lisp-user:bar
5
```

Кроме того, мы можем сделать еще один шаг и *импортировать* символ bar в текущий пакет, и тогда пакет mine будет делить его с пользовательским пакетом:

```
MINE> (import 'common-lisp-user:bar)
T
MINE> bar
5
```

На импортированный символ можно ссылаться без какого-либо префикса. Оба пакета теперь делят этот символ, а значит, уже не может быть отдельного символа mine:bar.

А что произойдет, если такой символ уже был? В таком случае импорт вызовет ошибку, подобную той, которую мы увидим, попытавшись импортировать sym:

```
MINE> (import 'common-lisp-user::sym)
Error: SYM is already present in MINE.
```

Ранее мы предпринимали неудачную попытку вычислить sym в mine, которая привела к тому, что этот символ был интернирован в mine. У него не было значения, и это вызвало ошибку, но интернирование все равно произошло, просто потому, что он был набран и считан, когда текущим пакетом был mine. Поэтому теперь, когда мы пытаемся импортировать sym в mine, символ с таким именем уже есть в этом пакете.

Другой способ получить доступ к символам другого пакета – *использовать* сам пакет:

```
MINE> (use-package 'common-lisp-user)
T
```

Теперь *все* символы, экспортированные пользовательским пакетом, принадлежат mine. (Если sym экспортировался пользовательским пакетом, то этот вызов приведет к ошибке.)

common-lisp – это пакет, содержащий имена всех встроенных операторов и функций. Так как мы передали имя этого пакета аргументу :use вызова make-package, который создал пакет mine, в нем будут видимы все имена Common Lisp:

```
MINE> #'cons
#<Compiled-Function CONS 462A3E>
```

Как и компиляция, операции с пакетами редко выполняются непосредственно в toplevel. Гораздо чаще их вызовы содержатся в файлах с исходным кодом. Обычно достаточно поместить в начале файла `defpackage` и `in-package`, как на стр. 148.

Модульность, создаваемая пакетами, несколько своеобразна. Она дает нам модули не объектов, а имен. Каждый пакет, использующий `common-lisp`, имеет доступ к символу `cons`, поскольку в пакете `common-lisp` задана функция с таким именем. Но как следствие этого и переменная с именем `cons` будет видима во всех пакетах, использующих `common-lisp`. Если пакеты сбивают вас с толку, то основная причина этого в том, что они основываются не на объектах, а на их именах.°

14.5. Loop

Макрос `loop` первоначально был разработан в помощь начинающим Лисп-программистам для написания итеративного кода, позволяя использовать для этого выражения, напоминающие обычные английские фразы, которые затем транслируются в Лисп-код. К несчастью, `loop` оказался похож на английский язык в большей степени, чем предполагали его создатели: в несложных случаях вы можете использовать его, совершенно не задумываясь о том, как это работает, но понять общий принцип его действия практически невозможно.

Если вы желаете ознакомиться с `loop` за один день, то для вас есть две новости: хорошая и плохая. Хорошая заключается в том, что вы не одиноки: мало кто в действительности понимает, как он работает. Плохая же состоит в том, что вы, вероятно, никогда этого не поймете, хотя бы потому, что стандарт ANSI фактически не предоставляет формального описания его поведения.

Единственным определением `loop` является его реализация, а единственным способом изучить его (насколько это возможно) являются примеры. В стандарте ANSI в главе, описывающей `loop`, имеется большой набор примеров, и мы применим здесь такой же подход, чтобы познакомить вас с основными концепциями этого макроса.

Первое, что люди замечают в макросе `loop`, – это наличие у него *синтаксиса*. Его тело состоит не из подвыражений, а из предложений, которые не разделяются скобками. Каждое предложение имеет свой синтаксис. В целом, `loop`-выражения напоминают Алгол-подобные языки. Но есть одно серьезное отличие, сильно отдаляющее `loop` от Алгола: порядок, в котором следуют предложения, весьма слабо определяет порядок, в котором они будут исполняться.

Вычисление `loop`-выражения осуществляется в три этапа, причем каждое предложение может вносить свой вклад более чем в один из них. Вот эти этапы:

1. *Пролог*. Выполняется один раз за весь вызов loop; устанавливает исходные значения переменных.
2. *Тело*. Вычисляется на каждой итерации. Начинается с проверок на завершение, следом за ними идут вычисляемые выражения, после которых обновляются итерируемые переменные.
3. *Эпилог*. Вычисляется по завершении всех итераций; определяет возвращаемое значение.

Рассмотрим некоторые примеры loop-выражений и прикинем, к каким этапам относятся их части. Один из простейших примеров:

```
> (loop for x from 0 to 9
        do (princ x))
0123456789
NIL
```

Данное выражение печатает целые числа от 0 до 9 и возвращает nil. Первое предложение:

```
for x from 0 to 9
```

вносит вклад в два первых этапа: устанавливает значение переменной x в 0 в прологе, сравнивает с 9 в начале тела и увеличивает переменную в конце. Второе предложение

```
do (princ x)
```

добавляет код (princ-выражение) в тело.

В более общем виде предложение с for определяет форму установления исходного значения и форму обновления. Завершение же может управляться чем-нибудь типа while или until:

```
> (loop for x = 8 then (/ x 2)
        until (< x 1)
        do (princ x))
8421
NIL
```

С помощью and вы можете создавать составные конструкции с for, содержащие несколько переменных и обновляющие их параллельно:

```
> (loop for x from 1 to 4
        and y from 1 to 4
        do (princ (list x y)))
(1 1) (2 2) (3 3) (4 4)
NIL
```

А если предложений с for будет несколько, то переменные будут обновляться поочередно.

Еще одной типичной задачей, решаемой во время итерации, является накопление каких-либо значений. Например:

```
> (loop for x in '(1 2 3 4)
        collect (1+ x))
(2 3 4 5)
```

При использовании `in` вместо `from` в `for`-предложении переменная по очереди принимает значения каждого из элементов списка, а не последовательных целых чисел.

В приведенном примере `collect` оказывает влияние на все три этапа. В прологе создается анонимный аккумулятор, имеющий значение `nil`; в теле `(1+ x)` к аккумулятору прибавляется 1, а в эпилоге возвращается значение аккумулятора.

Это наш первый пример, возвращающий какое-то конкретное значение. Существуют предложения для явного задания возвращаемого значения, но в их отсутствие этим занимается `collect`. Фактически эту же работу мы могли бы выполнить через `mapcar`.

Наиболее часто `loop` используется для сбора результатов вызова некоторой функции определенное количество раз:

```
> (loop for x from 1 to 5
        collect (random 10))
(3 8 6 5 0)
```

В данном примере мы получили список из пяти случайных чисел. Именно для этой цели мы уже определяли функцию `map-int` (стр. 117). Тогда зачем же мы это делали, если у нас есть `loop`? Но точно так же можно спросить: зачем нам нужен `loop`, когда у нас есть `map-int`?°

С помощью `collect` можно также аккумулировать значения в именованную переменную. Следующая функция принимает список чисел и возвращает списки его четных и нечетных элементов:

```
(defun even/odd (ns)
  (loop for n in ns
        if (evenp n)
           collect n into evens
           else collect n into odds
        finally (return (values evens odds))))
```

Предложение `finally` добавляет код в эпилог. В данном случае оно определяет возвращаемое выражение.

Предложение `sum` похоже на `collect`, но накапливает значения в виде числа, а не списка. Сумму всех чисел от `1` до `n` мы можем получить так:

```
(defun sum (n)
  (loop for x from 1 to n
        sum x))
```

Более детальное рассмотрение `loop` содержится в приложении D, начиная со стр. 331. Приведем несколько примеров: рис. 14.1 содержит две итеративные функции из предыдущих глав, а на рис. 14.2 показана аналогичная их реализация с помощью `loop`.

```
(defun most (fn lst)
  (if (null lst)
      (values nil nil)
      (let* ((wins (car lst))
             (max (funcall fn wins)))
        (dolist (obj (cdr lst))
          (let ((score (funcall fn obj)))
            (when (> score max)
              (setf wins obj
                    max  score))))
        (values wins max))))

(defun num-year (n)
  (if (< n 0)
      (do* ((y (- yzero 1) (- y 1))
            (d (- (year-days y)) (- d (year-days y))))
           ((<= d n) (values y (- n d))))
      (do* ((y yzero (+ y 1))
            (prev 0 d)
            (d (year-days y) (+ d (year-days y))))
           ((> d n) (values y (- n prev))))))
```

Рис. 14.1. *Итерация без loop*

Одно `loop`-предложение может ссылаться на переменные, устанавливаемые другим. Например, в определении `even/odd` предложение `finally` ссылается на переменные, установленные в двух предыдущих `collect`-предложениях. Отношения между такими переменными являются одним из самых неясных моментов при использовании `loop`. Сравните два выражения:

```
(loop for y = 0 then z
      for x from 1 to 5
      sum 1 into z
      finally (return (values y z)))

(loop for x from 1 to 5
      for y = 0 then z
      sum 1 into z
      finally (return (values y z)))
```

Они содержат всего четыре предложения и потому кажутся довольно простыми. Возвращают ли они одинаковые значения? Какие значения они вернут? Попытки найти ответ в стандарте языка окажутся тщетными. Каждое предложение внутри `loop` само по себе незатейливо, но *сочетаться* они могут довольно причудливым и в конечном итоге четко не заданным образом.

По этой причине использовать `loop` не рекомендуется. Самое лучшее, что можно сказать об этом макросе, – это то, что в типичных случаях,

как те, что изображены на рис. 14.2, он может сделать код более доступным для понимания.

```
(defun most (fn lst)
  (if (null lst)
      (values nil nil)
      (loop with wins = (car lst)
            with max = (funcall fn wins)
            for obj in (cdr lst)
            for score = (funcall fn obj)
            when (> score max)
                 do (setf wins obj
                          max score)
           finally (return (values wins max)))))

(defun num-year (n)
  (if (< n 0)
      (loop for y downfrom (- yzero 1)
            until (<= d n)
            sum (- (year-days y)) into d
            finally (return (values (+ y 1) (- n d))))
      (loop with prev = 0
            for y from yzero
            until (> d n)
            do (setf prev d)
            sum (year-days y) into d
            finally (return (values (- y 1)
                                    (- n prev))))))
```

Рис. 14.2. *Итерация с помощью loop*

14.6. Особые условия

В Common Lisp *особые условия* (*conditions*) включают ошибки и другие ситуации, которые могут возникать в процессе выполнения. Когда сигнализируется какое-либо условие, вызывается соответствующий обработчик. Обработчик по умолчанию для условий-ошибок обычно вызывает цикл прерывания. Но Common Lisp предоставляет и разнообразные операторы для сигнализации и обработки условий. Вы можете переопределять уже существующие обработчики и даже писать собственные.

Большинству программистов не придется работать с особыми условиями непосредственно. Однако есть несколько уровней более абстрактных операторов, которые используют условия, поэтому для их понимания все же полезно иметь представление о механизме, изложенном ниже.

В Common Lisp есть несколько операторов для сигнализации об ошибках. Основным является error. Один из способов его вызова – передача ему тех же аргументов, что и format:

```
> (error "Your report uses ~A as a verb." 'status)
Error: Your report uses STATUS as verb.
       Options: :abort, :backtrace
>>
```

Если такое условие не обрабатывается, выполнение будет прервано, как показано выше.

Более абстрактными операторами для сигнализации ошибок являются ecase, check-type и assert. Первый напоминает case, но сигнализирует об ошибке, когда не найдено совпадение ни с одним ключом:

```
> (ecase 1 (2 3) (4 5))
Error: No applicable clause.
       Options: :abort, :backtrace
>>
```

Обычное case-выражение в этом случае вернет nil, но, поскольку считается плохим стилем пользоваться этим возвращаемым значением, имеет смысл применить ecase, если у вас нет варианта otherwise.

Макрос check-type принимает место, имя типа и необязательный аргумент-строку. Он сигнализирует о *корректируемой ошибке*, если значение по данному месту не соответствует заданному типу. Обработчик корректируемой ошибки предложит один из вариантов ее исправления:

```
> (let ((x '(a b c)))
    (check-type (car x) integer "an integer")
    x)
Error: The value of (CAR X), A, should be an integer.
       Options: :abort, :backtrace, :continue
>> :continue
New value of (CAR X)? 99
(99 B C)
>
```

Только что мы привели пример коррекции ошибки, исправив значение (car x), после чего выполнение продолжилось с исправленным значением, как будто оно было передано таким изначально.

Этот макрос был определен с помощью более общего assert. Он принимает тестовое выражение и список одного или более мест, сопровождаемый теми же аргументами, которые вы бы передали в error:

```
> (let ((sandwich '(ham on rye)))
    (assert (eql (car sandwich) 'chicken)
            ((car sandwich))
            "I wanted a ~A sandwich." 'chicken)
    sandwich)
```

```
Error: I wanted a CHICHEN sandwich.
       Options: :abort, :backtrace, :continue
>> :continue
New value of (CAR SANDWICH)? 'chicken
(CHICKEN ON RYE)
>
```

Также имеется возможность создавать собственные обработчики, но этим редко кто пользуется. Обычно обходятся уже имеющимися средствами, например ignore-errors. Этот макрос ведет себя аналогично progn, если ни один из его аргументов не приводит к ошибке. Если же во время выполнения одного из выражений сигнализируется ошибка, то работа не прерывается, а выражение ignore-errors немедленно завершается с возвратом двух значений: nil и условия, которое было просигнализировано.

Например, если вы даете пользователю возможность вводить собственные выражения, но не хотите, чтобы синтаксически неверное выражение прерывало работу, то вам понадобится защита от некорректно сформированных выражений:

```
(defun user-input (prompt)
  (format t prompt)
  (let ((str (read-line)))
    (or (ignore-errors (read-from-string str))
        nil)))
```

Функция вернет nil, если считанное выражение содержит синтаксическую ошибку:

```
> (user-input "Please type an expression> ")
Please type an expression> #%@#+!!
NIL
```

15

Пример: логический вывод

В следующих трех главах вашему вниманию предлагаются три самостоятельные программы. Они были выбраны, чтобы показать, какой вид могут принимать программы на Лиспе, и продемонстрировать задачи, для решения которых этот язык особенно хорошо подходит.

В этой главе мы напишем программу, совершающую логические «умозаключения», основанные на наборе правил «если-то». Это классический пример, который не только часто встречается в учебниках, но и отражает изначальную концепцию Лиспа как языка «символьных вычислений». Многие ранние программы на Лиспе имеют черты той, которая описана ниже.

15.1. Цель

В данной программе мы собираемся представлять информацию в знакомой нам форме: списком из предиката и его аргументов. Представим факт отцовства Дональда по отношению к Нэнси таким образом:

```
(parent donald nancy)
```

Помимо фактов наша программа будет использовать правила, которые сообщают, что может быть выведено из имеющихся фактов. Мы будем представлять эти правила так:

(<- *заголовок тело*)

где *заголовок* соответствует следствию, а *тело* – условию. Переменные внутри них мы будем представлять символами, начинающимися со знака вопроса. Таким образом, следующее правило:

```
(<- (child ?x ?y) (parent ?y ?x))
```

сообщает, что если y является родителем x, то x является ребенком y, или, говоря более точно, мы можем доказать любой факт вида (child x y) через доказательство (parent y x).

Тело (условная часть правила) может быть составным выражением, включающим логические операторы and, or, not. Так, правило, согласно которому, если x является мужчиной и родителем y, то x – отец y, выглядит следующим образом:

```
(<- (father ?x ?y) (and (parent ?x ?y) (male ?x)))
```

Одни правила могут опираться на другие. К примеру, правило, определяющее, является ли x дочерью y, опирается на уже определенное правило родительства:

```
(<- (daughter ?x ?y) (and (child ?x ?y) (female ?x)))
```

При доказательстве выражений может применяться любое количество правил, необходимых для достижения твердой почвы фактов. Этот процесс иногда называют *обратной цепочкой логического вывода* (*backward chaining*). Обратной потому, что вывод сперва рассматривает часть-следствие, чтобы увидеть, будет ли правило полезным, прежде чем продолжать доказательство части-условия. А цепочкой он называется потому, что правила зависят друг от друга, формируя цепочку (скорее даже дерево) правил, которая ведет от того, что мы хотим доказать, к тому, что мы уже знаем.°

15.2. Сопоставление

Чтобы написать нашу программу для обратной цепочки логического вывода, нам понадобится функция, выполняющая сопоставление с образцом (pattern-matching). Эта функция должна сравнивать два списка, возможно, содержащие переменные, и проверять, есть ли такая комбинация значений переменных, при которой списки стали бы равными. Например, если ?x и ?y – переменные, то два списка:

```
(p ?x ?y c ?x)
(p  a  b c  a)
```

сопоставляются, если ?x = a и ?y = b, а списки

```
(p ?x b ?y a)
(p ?y b  c a)
```

сопоставляются при ?x = ?y = c.

На рис. 15.1 показана функция match, сопоставляющая два дерева. Если они могут совпадать, то она возвращает ассоциативный список, показывающий, каким образом они совпадают:

```
> (match '(p a b c a) '(p ?x ?y c ?x))
((?Y . B) (?X . A))
T
```

```
> (match '(p ?x b ?y a) '(p ?y b c a))
((?Y . C) (?X . ?Y))
T
> (match '(a b c) '(a a a)
NIL
```

По мере поэлементного сравнения match накапливает присваивания переменным значений, которые мы называем *связями* (*bindings*), в переменной binds. Если сопоставление завершилось успешно, то match возвращает сгенерированные связи, в противном случае возвращает nil. Так как не все успешные сопоставления генерируют хоть какие-то связи, match, по тому же принципу, что и gethash, возвращает второе значение, указывающее на успешность сопоставления.

```
> (match '(p ?x) '(p ?x))
NIL
T
```

```
(defun match (x y &optional binds)
 (cond
   ((eql x y) (values binds t))
   ((assoc x binds) (match (binding x binds) y binds))
   ((assoc y binds) (match x (binding y binds) binds))
   ((var? x) (values (cons (cons x y) binds) t))
   ((var? y) (values (cons (cons y x) binds) t))
   (t
    (when (and (consp x) (consp y))
      (multiple-value-bind (b2 yes)
                           (match (car x) (car y) binds)
        (and yes (match (cdr x) (cdr y) b2)))))))

(defun var? (x)
  (and (symbolp x)
       (eql (char (symbol-name x) 0) #\?)))

(defun binding (x binds)
  (let ((b (assoc x binds)))
    (if b
        (or (binding (cdr b) binds)
            (cdr b)))))
```

Рис. 15.1. *Функции для сопоставления*

Когда match возвращает nil и t, как произошло выше, это означает, что имеет место успешное сопоставление, не создавшее связей для переменных. Алгоритм действия match можно описать следующим образом:

1. Если x и y равны с точки зрения eql, то они сопоставляются; иначе:
2. Если x – это переменная, уже ассоциированная со значением, то она сопоставляется с y, если значения совпадают; иначе:

3. Если y – это переменная, уже ассоциированная со значением, то она сопоставляется с x, если значения совпадают; иначе:
4. Если переменная x не ассоциирована со значением, то с ней связывается текущее значение; иначе:
5. Если переменная y не ассоциирована со значением, то с ней связывается текущее значение; иначе:
6. Два значения сопоставляются, если они оба – cons-ячейки, их car-элементы сопоставляются, а cdr сопоставляются с учетом полученных связей.

Вот два примера, демонстрирующие по порядку все шесть случаев:

```
> (match '(p ?v b ?x d (?z ?z))
         '(p a ?w c ?y ( e  e))
         '((?v . a) (?w . b)))
((?Z . E) (?Y . D) (?X . C) (?V . A) (?W . B))
T
```

Чтобы найти ассоциированное значение (если оно имеется), match вызывает binding. Эта функция должна быть рекурсивной, так как в результате сопоставления могут получиться пары, в которых переменная связана со значением косвенно, например ?x может быть связан с a через список, содержащий (?x . ?y) и (?y . a).

```
> (match '(?x a) '(?y ?y))
((?Y . A) (?X . ?Y))
T
```

Сопоставив ?x с ?y, а ?y с a, мы видим косвенную связь переменной ?x со значением.

15.3. Отвечая на запросы

Теперь, когда введены основные концепции сопоставления, мы можем перейти непосредственно к назначению нашей программы: если мы имеем выражение, вероятно, содержащее переменные, то исходя из имеющихся фактов и правил мы можем найти все ассоциации, делающие это выражение истинным. Например, если у нас есть только тот факт, что

```
(parent donald nancy)
```

и мы просим программу доказать

```
(parent ?x ?y)
```

то она вернет нечто, похожее на

```
(((?x . donald) (?y . nancy)))
```

Это означает, что наше выражение может быть истинным лишь в одном случае: ?x – это donald, а ?y – nancy.

Поскольку у нас есть функция сопоставления, можно утверждать, что мы на правильном пути. Теперь нам нужно научиться определять правила. Соответствующий код приведен на рис. 15.2. Правила содержатся в хеш-таблице `*rules*` в соответствии с предикатами в заголовках. Это вводит ограничение, что переменные не могут находиться на месте предикатов. От него можно избавиться, если хранить такие правила в отдельном списке, но тогда для доказательства чего-либо нам пришлось бы сопоставлять друг другу каждое правило из этого списка.

```
(defvar *rules* (make-hash-table))

(defmacro <- (con &optional ant)
  `(length (push (cons (cdr ',con) ',ant)
                 (gethash (car ',con) *rules*))))
```

Рис. 15.2. *Определение правил*

Мы будем использовать макрос `<-` для определения и правил, и фактов. Факт представляется в виде правила, содержащего только заголовок. Это соответствует нашему представлению о правилах. Правило гласит, что для доказательства заголовка необходимо доказать тело, а раз тела нет, то и доказывать нечего. Вот два уже знакомых нам примера:

```
> (<- (parent donald nancy))
1
> (<- (child ?x ?y) (parent ?y ?x))
1
```

Вызов `<-` возвращает номер нового правила для заданного предиката; оборачивание `length` вокруг `push` позволяет избежать большого объема информации, выводимой в toplevel.

На рис. 15.3 содержится большая часть кода, нужного нам для вывода.

Функция `prove` является осью, вокруг которой вращается весь логический вывод. Она принимает выражение и необязательный набор связей. Если выражение не содержит логических операторов, то вызывается `prove-simple`. Именно здесь происходит сам обратный вывод. Эта функция ищет все правила с истинным предикатом и пытается сопоставить заголовок каждого из них с фактом, который мы хотим доказать. Затем для каждого совпавшего заголовка доказывается его тело с учетом новых связей, созданных `match`. Вызовы `prove` возвращают списки связей, которые затем собираются `mapcan`:

```
> (prove-simple 'parent '(donald nancy) nil)
(NIL)
> (prove-simple 'child '(?x ?y) nil)
(((#:?6 . NANCY) (#:?5 . DONALD) (?Y . #:?5) (?X . #:?6)))
```

```
(defun prove (expr &optional binds)
  (case (car expr)
    (and (prove-and (reverse (cdr expr)) binds))
    (or  (prove-or (cdr expr) binds))
    (not (prove-not (cadr expr) binds))
    (t   (prove-simple (car expr) (cdr expr) binds))))

(defun prove-simple (pred args binds)
  (mapcan #'(lambda (r)
              (multiple-value-bind (b2 yes)
                                   (match args (car r) binds)
                (when yes
                  (if (cdr r)
                      (prove (cdr r) b2)
                      (list b2)))))
          (mapcar #'change-vars
                  (gethash pred *rules*))))

(defun change-vars (r)
  (sublis (mapcar #'(lambda (v) (cons v (gensym "?")))
                  (vars-in r))
          r))

(defun vars-in (expr)
  (if (atom expr)
      (if (var? expr) (list expr))
      (union (vars-in (car expr))
             (vars-in (cdr expr)))))
```

Рис. 15.3. *Вывод*

Оба полученных значения подтверждают, что есть лишь один путь доказательства. (Если доказать утверждение не получилось, возвращается `nil`.) Первый пример был выполнен без создания каких-либо связей, в то время как во втором примере переменные `?x` и `?y` были связаны (косвенно) с `nancy` и `donald`.

Между прочим, мы видим здесь хороший пример высказанной ранее (стр. 39) идеи: поскольку наша программа написана в функциональном стиле, мы можем тестировать каждую функцию отдельно.

Теперь пара слов о том, зачем нужен `gensym`. Раз мы собираемся использовать правила, содержащие переменные, то нам нужно избегать наличия двух правил с одной и той же переменной. Рассмотрим два правила:

```
(<- (child ?x ?y) (parent ?y ?x))

(<- (daughter ?y ?x) (and (child ?y ?x) (female ?y)))
```

В них утверждается, что для *любого* x и y 1) x является ребенком y, если y является родителем x; 2) y является дочкой x, если y является ребенком x и y – женщина. Связь между переменными в рамках одного пра-

вила имеет большое значение, а вот факт, что два правила используют одинаковые имена для определения переменных, совершенно случаен.

Если мы воспользуемся этими правилами в том виде, в каком только что их записали, то они не будут работать. Если мы попытаемся доказать, что a – дочь b, то сопоставление с заголовком второго правила даст связи ?y = a и ?x = b. Имея такие связи, мы не сможем воспользоваться первым правилом:

```
> (match '(child ?y ?x)
         '(child ?x ?y)
         '((?y . a) (?x . b)))
NIL
```

Для гарантии того, что переменные будут действовать лишь в пределах одного правила, мы заменяем все переменные внутри правила на gensym. Для этой цели определена функция change-vars. Так мы приобретаем защиту от совпадений с другими переменными в определениях других правил. Но, поскольку правила могут применяться рекурсивно, нам также нужна защита при сопоставлении с этим же правилом. Для этого change-vars должна вызываться не только при определении правил, но и при каждом их использовании.

Теперь остается лишь определить функции для доказательства составных утверждений. Соответствующий код приведен на рис. 15.4. Обработка выражений or или not предельно проста. В первом случае мы собираем все связи, полученные из каждого выражения в or. Во втором случае мы просто возвращаем имеющиеся связи, если выражение внутри not возвратило nil.

```
(defun prove-and (clauses binds)
  (if (null clauses)
      (list binds)
      (mapcan #'(lambda (b)
                  (prove (car clauses) b))
              (prove-and (cdr clauses) binds))))

(defun prove-or (clauses binds)
  (mapcan #'(lambda (c) (prove c binds))
          clauses))

(defun prove-not (clause binds)
  (unless (prove clause binds)
    (list binds)))
```

Рис. 15.4. *Логические операторы*

Функция prove-and лишь чуть сложнее. Она работает как фильтр, доказывая первое выражение для каждого из наборов связей, полученных из остальных выражений. По этой причине выражения внутри and рас-

сматриваются в обратном порядке. Переворачивание результата prove-and компенсирует это явление (так как результирующий список выражений формируется функцией в обратном порядке, в конце ее выполнения этот список нужно развернуть).

Теперь у нас есть рабочая программа, но она не очень удобна для конечного пользователя. Разбирать списки связей, возвращаемых prove, довольно сложно, а ведь с усложнением выражений они будут только расти. Эту проблему решает макрос with-answer, изображенный на рис. 15.5. Он принимает запрос (не вычисленный) и вычисляет свое тело для каждого набора связей, полученных при обработке запроса, связывая каждую переменную запроса со значением, которое она имеет в текущем экземпляре связей:

```
> (with-answer (parent ?x ?y)
    (format t "~A is the parent of ~A.~%" ?x ?y))
DONALD is the parent of NANCY.
NIL
```

Этот макрос расшифровывает полученные связи и предоставляет удобный способ использования prove в наших программах. На рис. 15.6 показан результат его раскрытия, а рис. 15.7 демонстрирует некоторые примеры использования этого макроса.

```
(defmacro with-answer (query &body body)
  (let ((binds (gensym)))
    `(dolist (,binds (prove ',query))
       (let ,(mapcar #'(lambda (v)
                         `(,v (binding ',v ,binds)))
                     (vars-in query))
         ,@body))))
```

Рис. 15.5. *Интерфейсный макрос*

```
(with-answer (p ?x ?y)
  (f ?x ?y))

;;; раскрывается в:

(dolist (#:g1 (prove '(p ?x ?y)))
  (let ((?x (binding '?x #:g1))
        (?y (binding '?y #:g1)))
    (f ?x ?y)))
```

Рис. 15.6. *Раскрытие вызова with-answer*

Если мы выполним (clrhash *rules*) и затем определим следующие правила и факты:

```
(<- (parent donald nancy))
(<- (parent donald debbie))
(<- (male donald))
(<- (father ?x ?y) (and (parent ?x ?y) (male ?x)))
(<- (= ?x ?y))
(<- (sibling ?x ?y) (and (parent ?z ?x)
                         (parent ?z ?y)
                         (not (= ?x ?y))))
```

то сможем сделать следующие выводы:

```
> (with-answer (father ?x ?y)
    (format t "~A is the father of ~A.~%" ?x ?y))
DONALD is the father of DEBBIE.
DONALD is the father of NANCY.
NIL
> (with-answer (sibling ?x ?y)
    (format t "~A is the sibling of ~A.~%" ?x ?y))
DEBBIE is the sibling of NANCY.
NANCY is the sibling of DEBBIE.
NIL
```

Рис. 15.7. *Программа в работе*

15.4. Анализ

Может показаться, что написанный нами код является простым и естественным решением поставленной задачи. На самом деле, он крайне неэффективен. В действительности, мы написали, по сути, интерпретатор, в то время как могли создать компилятор.

Приведем набросок того, как это может быть сделано. Основная идея заключается в запаковке всей программы в макросы <- и with-answer. В таком случае основная часть работы будет выполняться на этапе компиляции, тогда как сейчас она выполняется непосредственно во время запуска. (Зародыш этой идеи можно увидеть в avg на стр. 181.) Будем представлять правила как функции, а не как списки. Вместо функций типа prove и prove-and, интерпретирующих выражения в процессе работы, у нас будут функции для преобразования выражений в код. Выражения становятся доступны в момент определения правила. Зачем ждать, пока выражение будет использовано, чтобы его проанализировать? Это относится и к with-answer, который будет вызывать функции типа <- для генерации своего раскрытия.

Кажется, что подобная программа будет значительно сложнее написанной в этой главе, но на деле реализация предложенной идеи займет всего в два-три раза больше времени. Читателям, желающим узнать больше о подобных методиках, рекомендуется посмотреть книги «On Lisp» и «Paradigms of Artificial Intelligence Programming»[1], которые содержат примеры программ, написанных в таком стиле.

[1] Peter Norvig «Paradigms of Artificial Intelligence Programming», Morgan Kaufman, 1992. Эта книга, также известная как PAIP, рассматривает программирование задач искусственного интеллекта. Автор использует Common Lisp и сопровождает книгу большим количеством кода, который доступен по адресу: *http://norvig.com/paip/README.html*. – *Прим. перев.*

16

Пример: генерация HTML

В этой главе мы напишем небольшой HTML-генератор – программу, автоматически производящую набор связанных веб-страниц. Она не только демонстрирует различные концепции Лиспа, но и является хорошим примером разработки «снизу-вверх». Мы начнем с создания HTML-утилит общего назначения, которые затем будем рассматривать как язык, на котором и будем писать собственно генератор.

16.1. HTML

HTML (HyperText Markup Language, язык разметки гипертекста) – это то, из чего состоят веб-страницы. Это очень простой язык, не имеющий продвинутых возможностей, но зато он легок в изучении. В этом разделе представлен обзор HTML.

Веб-страницы просматриваются с помощью веб-браузера, который получает код страницы (как правило, с удаленного компьютера), преобразовывает его в читаемый вид и выводит на экран. HTML-файл – это текстовый файл, содержащий *разметку*, воспринимаемую браузером как инструкции.

На рис. 16.1 приведен пример простого HTML-файла, а на рис. 16.2 показано, как он будет отображаться в браузере. Обратите внимание, что текст между угловыми скобками не отображается. Это и есть разметка. HTML имеет два вида меток. Одни используются попарно:

```
<метка> ... </метка>
```

Первая метка обозначает начало некоторого окружения, вторая помечает его окончание. Одной из меток такого рода является <h2>. Весь текст, находящийся между <h2> и </h2>, отображается увеличенным шрифтом. (Наибольший шрифт задается <h1>.)

```
<center>
<h2>Your Fortune</h2>
</center>
<br><br>
Welcome to the home page of the Fortune Cookie
Institute. FCI is a non-profit institution
dedicated to the development of more realistic
fortunes. Here are some examples of fortunes
that fall within our guidelines:
<ol>
<li>Your nostril hairs will grow longer.
<li>You will never learn how to dress properly.
<li>Your car will be stolen.
<li>You will gain weight.
</ol>
Click <a href="research.html">here</a> to learn
more about our ongoing research projects.
```

Рис. 16.1. *HTML-файл*

Your Fortune

Welcome to the home page of the Fortune Cookie Institute. FCI is a nonprofit institution dedicated to the development of more realistic fortunes. Here are some examples of fortunes that fall within our guidelines:

1. Your nostril hairs will grow longer.
2. You will never learn how to dress properly.
3. Your car will be stolen.
4. You will gain weight.

Click **here** to learn more about our ongoing research projects.

Рис. 16.2. *Внешний вид веб-страницы*

Другие парные метки: `<ol>` («ordered list», упорядоченный список) для создания нумерованного списка; `<center>` для центрирования текста; `<a...>` («anchor», якорь) для создания ссылки.

Именно ссылки превращают текст в гипертекст. Текст, находящийся между `<a...>` и `</a>`, отображается браузерами особым образом, как правило, с подчеркиванием. При нажатии на ссылку осуществляется переход на другую страницу, адрес которой содержится внутри метки. Следующая метка

```
<a href="foo.html">
```

означает ссылку на другой HTML-файл, находящийся в том же каталоге. Таким образом, при нажатии на ссылку на рис. 16.2 браузер загрузит и отобразит файл "research.html".

Ссылки не обязаны указывать на файлы в одном и том же каталоге, но могут указывать и на любой адрес в Интернете (хотя в нашем примере это не используется).

Другая разновидность меток не имеет маркера окончания. В нашем примере (см. рис. 16.1) используются метки
 («break», разрыв) для перехода на новую строку и <li> («list item», элемент списка) для обозначения элемента внутри окружения списка. Разумеется, HTML содержит и другие метки, но в этой главе нам понадобятся лишь те, которые упомянуты на рис. 16.1.

16.2. Утилиты HTML

В этом разделе мы определим некоторые утилиты для генерации HTML. На рис. 16.3 показаны базовые утилиты для генерации разметки. Все они направляют вывод в *standard-output*, но мы всегда сможем перенаправить его, переназначив эту переменную.

```
(defmacro as (tag content)
  `(format t "<~(~A~)>~A</~(~A~)>"
           ',tag ,content ',tag))

(defmacro with (tag &rest body)
  `(progn
    (format t "~&<~(~A~)>~%" ',tag)
    ,@body
    (format t "~&<~(~A~)>~%" ',tag)))

(defun brs (&optional (n 1))
  (fresh-line)
  (dotimes (i n)
    (princ "<br>"))
  (terpri))
```

Рис. 16.3. *Утилиты для генерации разметки*

Макросы as и with предназначены для генерации выражений, окруженных парой меток. Первый принимает строку и печатает ее между метками:

```
> (as center "The Missing Lambda")
<center>The Missing Lambda</center>
NIL
```

Второй принимает тело кода и выводит результат его выполнения между метками:

```
> (with center
    (princ "The Unbalanced Parenthesis"))
<center>
The Unbalanced Parenthesis
</center>
NIL
```

Оба эти макроса используют управляющие директивы ~(и ~) для печати меток в нижнем регистре. На самом деле, HTML не чувствителен к регистру, но метки в нижнем регистре легче воспринимаются, особенно если их много.

В то время как макрос as пытается разместить весь вывод на одной строке, макрос with располагает метки на отдельных строках. (Директива ~& обеспечивает начало вывода с новой строки.) Это делается лишь для того, чтобы HTML-файлы лучше читались. При отображении страниц пробельные символы вокруг меток не показываются.

Последняя утилита на рис. 16.3, brs, генерирует множественные разрывы строк. Во многих браузерах они могут использоваться для управления вертикальными отступами.

Рисунок 16.4 содержит утилиты для непосредственной генерации HTML. Первая возвращает имя файла по заданному символу. В реальном приложении она могла бы вернуть полный путь к файлу в указанном каталоге. Здесь же она просто присоединяет к имени расширение ".html".

Макрос page служит для генерации всей страницы. Он схож с with-open-file, на основе которого построен. Выражения в его теле будут выполнены с привязкой *standard-output* к потоку, созданному при открытии файла, соответствующего символу name.

```
(defun html-file (base)
   (format nil "~(~A~).html" base))

(defmacro page (name title &rest body)
  (let ((ti (gensym)))
    `(with-open-file (*standard-output*
                      (html-file ,name)
                      :direction :output
                      :if-exists :supersede)
       (let ((,ti ,title))
         (as title ,ti)
         (with center
           (as h2 (string-upcase ,ti)))
         (brs 3)
         ,@body))))
```

Рис. 16.4. *Утилиты создания файлов*

В разделе 6.7 было показано, как присваивать временные значения специальным переменным. В примере на стр. 124 мы устанавливали значение *print-base* в 16 в теле let. Раскрытие page похожим образом связывает *standard-output* с потоком, указывающим на HTML-файл. Если мы вызовем as или princ в теле page, то вывод будет перенаправлен в соответствующий файл.

Заголовок title будет напечатан в самом верху страницы. За ним следует весь остальной вывод. Таким образом, вызов:

```
(page 'paren "The Unbalanced Parenthesis"
  (princ "Something in his expression told her..."))
```

приведет к тому, что файл "paren.html" будет иметь (поскольку html-file уже определен) следующее содержание:

```
<title>The Unbalanced Parenthesis</title>
<center>
<h2>THE UNBALANCED PARENTHESIS</h2>
</center>
<br><br><br>
Something in his expression told her...
```

Здесь нам знакомы все метки, кроме <title>. Текст, заключенный в метку <title>, не будет появляться нигде на странице, но зато он будет отображаться браузером в заголовке окна.

```
(defmacro with-link (dest &rest body)
  `(progn
     (format t "<a href=\"~A\">" (html-file ,dest))
     ,@body
     (princ "</a>")))

(defun link-item (dest text)
  (princ "<li>")
  (with-link dest
    (princ text)))

(defun button (dest text)
  (princ "[ ")
  (with-link dest
    (princ text))
  (format t " ]~%"))
```

Рис. 16.5. *Утилиты для генерации ссылок*

На рис. 16.5 представлены утилиты для генерации ссылок. Макрос with-link похож на with. Он принимает тело кода и вычисляет его между выражениями, которые генерируют ссылку на HTML-файл, имя которого было передано вторым аргументом:

```
> (with-link 'capture
    (princ "The Captured Variable"))
<a href="capture.html">The Captured Variable</a>
"</a>"
```

Он используется в функции link-item, которая принимает строку и создает элемент списка, также являющийся ссылкой,

```
> (link-item 'bq "Backquote!")
<li><a href="bq.html"> Backquote!</a>
"</a>"
```

и в функции button, которая генерирует ссылку внутри квадратных скобок,

```
> (button 'help "Help")
[ <a href="help.html">Help</a> ]
NIL
```

так что она напоминает кнопку.

16.3. Утилита для итерации

В данном разделе мы определим утилиты общего назначения, которые нам понадобятся в дальнейшем. Можем ли мы заранее знать, что в процессе разработки нам понадобится та или иная утилита? Нет, не можем. Обычно в процессе написания программы вы осознаете необходимость какой-либо утилиты, останавливаетесь, чтобы создать ее, и затем продолжаете писать программу, используя эту утилиту. Но описание здесь реального процесса разработки со всеми его остановками и возвратами было бы весьма запутанным. Поэтому мы ограничимся лишь конечным результатом и оговоркой, что написание программ никогда не будет столь прямолинейным, как это описано в книге. Оно всегда будет включать неоднократное полное или частичное переписывание кода.

Наша новая утилита (рис. 16.6) будет разновидностью mapc. Она принимает функцию трех аргументов и список. Для каждого элемента списка функция применяется как для самого элемента, так и для предшествующего и последующего элементов. (Если предшествующего или последующего элемента нет, то вместо них используется nil.)

```
> (map3 #'(lambda (&rest args) (princ args))
        '(a b c d))
(A NIL B) (B A C) (C B D) (D C NIL)
NIL
```

Как и mapc, она всегда возвращает nil.[1] Ситуации, в которых требуется подобная утилита, возникают довольно часто. С одной из них мы столк-

[1] На самом деле, mapc возвращает значение своего второго аргумента. – *Прим. перев.*

немся в следующем разделе, где нам понадобятся ссылки на предыдущую и следующую страницы.

```
(defun map3 (fn lst)
  (labels ((rec (curr prev next left)
             (funcall fn curr prev next)
             (when left
               (rec (car left)
                    curr
                    (cadr left)
                    (cdr left)))))
    (when lst
      (rec (car lst) nil (cadr lst) (cdr lst)))))
```

Рис. 16.6. *Итерация по тройкам*

Вариантом общей проблемы может быть случай, когда вам нужно сделать что-либо между каждой парой элементов списка:

```
> (map3 #'(lambda (c p n)
            (princ c)
            (if n (princ " | ")))
        '(a b c d))
A | B | C | D
NIL
```

Программисты периодически сталкиваются с подобной проблемой – пожалуй, не настолько часто, чтобы иметь встроенный оператор языка для ее решения, но достаточно часто для того, чтобы иметь возможность определить такой оператор самостоятельно.

16.4. Генерация страниц

Подобно книгам и журналам, наборы веб-страниц часто имеют древовидную организацию. Книга может состоять из глав, которые содержат разделы, которые, в свою очередь, состоят из подразделов, и т. д. Веб-страницы часто имеют подобную структуру, хотя и не используют подобные термины.

В этом разделе мы напишем программу, генерирующую набор веб-страниц, имеющих следующую структуру. Первая страница представляет собой оглавление, содержащее ссылки на *разделы*, которые, в свою очередь, содержат ссылки на *пункты*, являющиеся отдельными страницами с обычным текстом. Каждая страница должна иметь ссылки для перемещения назад, вперед и наверх по древовидной структуре. Ссылки вперед и назад осуществляют переход в пределах одного уровня вложенности. Например, ссылка вперед на странице, представляющей пункт, будет указывать на следующий пункт данного раздела. По ссылке вверх

осуществляется переход от пункта к разделу, а от раздела к оглавлению. Кроме того, должен быть индекс – отдельная страница со ссылками, на которой перечисляются все пункты в алфавитном порядке. Описанная иерархия изображена на рис. 16.7.

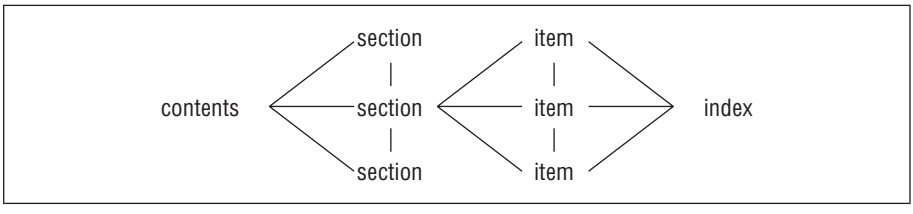

Рис. 16.7. *Структура сайта*

Операторы и структуры данных, необходимые для создания страниц, представлены на рис. 16.8. Наша программа будет работать с двумя типами объектов: пунктами, содержащими блоки текста, и разделами, содержащими списки ссылок на пункты.

```
(defparameter *sections* nil)

(defstruct item
  id title text)

(defstruct section
  id title items)

(defmacro defitem (id title text)
  `(setf ,id
         (make-item :id     ',id
                    :title  ,title
                    :text   ,text)))

(defmacro defsection (id title &rest items)
  `(setf ,id
         (make-section :id    ',id
                       :title ,title
                       :items (list ,@items))))

(defun defsite (&rest sections)
  (setf *sections* sections))
```

Рис. 16.8. *Создание сайта*

И разделы, и пункты содержат поле `id`. В качестве содержимого этого поля будут передаваться символы, имеющие два применения. Первое заключается в определениях `defitem` и `defsection`; здесь `id` – это уникальное имя, по которому мы будем ссылаться на пункт или раздел. Второе

применение: мы будем использовать id в имени файла, представляющего пункт или раздел. К примеру, страница, представляющая пункт foo, будет записана в файл "foo.html".

Кроме того, разделы и пункты имеют поля title, которые должны быть строками и будут использоваться в качестве заголовков соответствующих страниц.

Порядок следования пунктов в разделе определяется из аргументов defsection, порядок следования разделов в оглавлении – из аргументов defsite.

На рис. 16.9. представлены функции, генерирующие оглавление и индекс. Константы contents и index являются строками, служащими заголовками этих страниц и именами соответствующих файлов.

```
(defconstant contents "contents")
(defconstant index    "index")

(defun gen-contents (&optional (sections *sections*))
  (page contents contents
    (with ol
      (dolist (s sections)
        (link-item (section-id s) (section-title s))
        (brs 2))
      (link-item index (string-capitalize index)))))

(defun gen-index (&optional (sections *sections*))
  (page index index
    (with ol
      (dolist (i (all-items sections))
        (link-item (item-id i) (item-title i))
        (brs 2)))))

(defun all-items (sections)
  (let ((is nil))
    (dolist (s sections)
      (dolist (i (section-items s))
        (setf is (merge 'list (list i) is #'title<))))
    is))

(defun title< (x y)
  (string-lessp (item-title x) (item-title y)))
```

Рис. 16.9. *Генерация индекса и оглавления*

Функции gen-contents и gen-index в целом похожи друг на друга. Каждая из них открывает HTML-файл, генерирует заголовок и список ссылок. Разница лишь в том, что список пунктов в индексе должен быть отсортирован. Он строится с помощью функции all-items, которая использует функцию упорядочения title<. Важно, что все заголовки срав-

ниваются функцией string-lessp, которая, в отличие от string<, игнорирует регистр знаков.

В реальном приложении сравнение может быть более утонченным, например, оно может игнорировать начальные артикли «a» и «the».

Оставшийся код показан на рис. 16.10: gen-site генерирует весь набор веб-страниц, остальные функции генерируют разделы и пункты.

```
(defun gen-site ()
  (map3 #'gen-section *sections*)
  (gen-contents)
  (gen-index))

(defun gen-section (sect <sect sect>)
  (page (section-id sect) (section-title sect)
    (with ol
      (map3 #'(lambda (item <item item>)
                (link-item (item-id item)
                           (item-title item))
                (brs 2)
                (gen-item sect item <item item>))
            (section-items sect)))
    (brs 3)
    (gen-move-buttons (if <sect (section-id <sect))
                      contents
                      (if sect> (section-id sect>)))))

(defun gen-item (sect item <item item>)
  (page (item-id item) (item-title item)
    (princ (item-text item))
    (brs 3)
    (gen-move-buttons (if <item (item-id <item))
                      (section-id sect)
                      (if item> (item-id item>)))))

(defun gen-move-buttons (back up forward)
  (if back (button back "Back"))
  (if up (button up "Up"))
  (if forward (button forward "Forward")))
```

Рис. 16.10. *Генерация сайта, разделов и пунктов*

Под полным набором страниц понимается оглавление, индекс, страницы, представляющие каждый из разделов, и страницы, представляющие каждый из пунктов. Оглавление и индекс создаются с помощью функций, представленных на рис. 16.9. Разделы и пункты генерируются gen-section, которая создает страницу раздела и вызывает gen-item для генерации страниц, представляющих каждый пункт в данном разделе.

Эти две функции начинаются и заканчиваются похожим образом: обе принимают аргументы, представляющие сам объект, а также предыдущий и последующий объекты того же уровня вложенности; обе заканчиваются вызовом gen-move-buttons с целью сгенерировать кнопки для перемещения вперед, назад и вверх. Разница в середине: в отличие от gen-section, которая производит упорядоченный список ссылок на пункты, gen-item просто отправляет текст в выходной файл.

Каким будет текст каждого из пунктов – личное дело пользователя. Этот текст, к примеру, вполне может содержать HTML-разметку. Или же он может быть сгенерирован другой программой.

Пример ручного создания небольшого набора страниц приведен на рис. 16.11. В нем приводится перечень недавних публикаций в Институте Печенюшек Судьбы.

```
(defitem des "Fortune Cookies: Dessert or Fraud?" "...")

(defitem case "The Case for Pessimism" "...")

(defsection position "Position Papers" des case)

(defitem luck "Distribution of Bad Luck" "...")

(defitem haz "Health Hazards of Optimism" "...")

(defsection abstract "Research Abstracts" luck haz)

(defsite position abstract)
```

Рис. 16.11. *Небольшой сайт*

17

Пример: объекты

В этой главе мы собираемся реализовать внутри Лиспа свой собственный объектно-ориентированный язык. Такие программы называют *встроенными языками* (*embedded language*). Встраивание объектно-ориентированного языка в Лисп – это идеальный пример. Он не только демонстрирует типичное использование Лиспа, но еще и показывает, как естественно ложатся основные понятия объектно-ориентированного программирования на фундаментальные абстракции Лиспа.

17.1. Наследование

В разделе 11.10 объяснялось, чем обобщенные функции отличаются от передачи сообщений. В модели передачи сообщений объекты:

1. Имеют свойства.
2. Реагируют на сообщения.
3. Наследуют свойства и методы от предков.

CLOS, как известно, реализует модель обобщенных функций. Но в этой главе мы заинтересованы в написании простейшей объектной системы, не претендующей на соперничество с CLOS, поэтому будем использовать более старую модель.

Лисп предоставляет несколько способов хранения наборов свойств. Первый способ – представление объекта как хеш-таблицы и хранение его свойств в качестве ее элементов. В этом случае мы можем получить доступ к конкретным свойствам с помощью `gethash`:

```
(gethash 'color obj)
```

Поскольку функции – это объекты данных, мы также можем хранить их как свойства. Это означает, что мы также можем хранить и методы.

Чтобы вызвать определенный метод объекта, необходимо применить funcall к свойству с именем метода:

```
(funcall (gethash 'move obj) obj 10)
```

Используя такой подход, можно определить синтаксис передачи сообщений в стиле Smalltalk:

```
(defun tell (obj message &rest args)
  (apply (gethash message obj) obj args))
```

Теперь, чтобы приказать объекту передвинуться на 10, мы можем сказать:

```
(tell obj 'move 10)
```

По сути, единственный ингредиент, которого не хватает голому Лиспу, – это поддержка наследования. Мы можем реализовать простейший ее вариант с помощью определения рекурсивной версии gethash, как показано на рис. 17.1. (Имя rget – это сокращение от «recursive get», рекурсивное получение.) Теперь с помощью восьми строк кода мы получаем все три вышеперечисленные элемента объектной ориентированности.

```
(defun rget (prop obj)
  (multiple-value-bind (val in) (gethash prop obj)
    (if in
        (values val in)
        (let ((par (gethash :parent obj)))
          (and par (rget prop par))))))

(defun tell (obj message &rest args)
  (apply (rget message obj) obj args))
```

Рис. 17.1. *Наследование*

Давайте опробуем этот код в деле. Применим его к рассмотренному ранее примеру, создав два объекта, один из которых является потомком другого:

```
> (setf circle-class                 (make-hash-table)
        our-circle                   (make-hash-table)
        (gethash :parent our-circle) circle-class
        (gethash 'radius our-circle) 2)
2
```

Объект circle-class будет содержать метод area для всех кругов. Это будет функция одного аргумента – объекта, которому посылается сообщение:

```
> (setf (gethash 'area circle-class)
        #'(lambda (x)
            (* pi (expt (rget 'radius x) 2))))
#<Interpreted-Function BF1EF6>
```

Теперь можно запросить площадь our-circle. Она будет вычисляться в соответствии с только что определенным для класса методом: rget считывает свойство, а tell применяет метод:

```
> (rget 'radius our-circle)
2
T
> (tell our-circle 'area)
12.566370614359173
```

Прежде чем браться за усовершенствование данной программы, разберемся с тем, что мы уже сделали. Те самые восемь строчек кода превратили голую версию Лиспа без CLOS в объектно-ориентированный язык. Как нам удалось решить столь непростую задачу? Должно быть, тут скрыт какой-то трюк: реализовать объектную ориентированность восемью строчками кода!

Да, трюк здесь определенно присутствует, но он скрыт не в нашей программе. Дело в том, что Лисп в некотором смысле уже был объектно-ориентированным или скорее даже более общим языком. Все, что нам понадобилось, – это добавить новый фасад вокруг абстракций, которые в нем уже имелись.

17.2. Множественное наследование

Пока что мы умеем осуществлять лишь одиночное наследование, при котором объект может иметь только одного родителя. Но можно реализовать и множественное наследование, сделав parent списком и переопределив rget, как показано на рис. 17.2.

```
(defun rget (prop obj)
  (dolist (c (precedence obj))
    (multiple-value-bind (val in) (gethash prop c)
      (if in (return (values val in))))))

(defun precedence (obj)
  (labels ((traverse (x)
             (cons x
                   (mapcan #'traverse
                           (gethash :parents x)))))
    (delete-duplicates (traverse obj))))
```

Рис. 17.2. *Множественное наследование*

При одиночном наследовании, если мы хотим получить некоторое свойство объекта, нам достаточно рекурсивно пройтись по всем его предшественникам. Если сам объект не содержит достаточной информации об интересующем нас свойстве, мы переходим к его родителю, и т. д. При

подобном поиске в случае множественного наследования наша работа несколько усложняется, потому что предшественники объекта образуют не простое дерево, а граф, и обычный поиск в глубину здесь не поможет. Множественное наследование позволяет реализовать, например, такую иерархию, как на рис. 17.3: к `a` мы можем спуститься от `b` и `c`, а к ним – от `d`. Поиск в глубину (здесь скорее в высоту) даст следующий порядок обхода: `a`, `b`, `d`, `c`, `d`. Если желаемое свойство имеется в `d` и `c`, мы получим значение из `d`, а не из `c`, что не будет соответствовать правилу, по которому значение из подкласса приоритетнее, чем из его родителей.

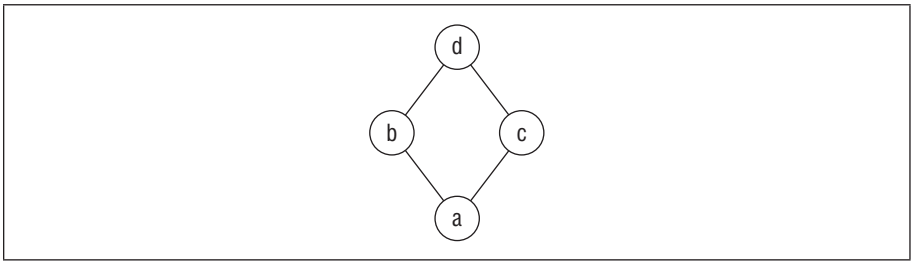

Рис. 17.3. *Несколько путей к одному суперклассу*

Чтобы соблюсти это правило, нам необходимо четко следить, чтобы никакой объект не обрабатывался раньше всех его потомков. В нашем случае порядок обхода должен быть таким: `a`, `b`, `c`, `d`. Как это гарантировать? Простейший способ – собрать список из объекта и всех его предков, следующих в правильном порядке, а затем проверять по очереди каждый элемент этого списка.

Такой список возвращает функция `precedence`. Она начинается с вызова `traverse`, выполняющего поиск в глубину. Если несколько объектов имеют общего предка, то он встретится в списке несколько раз. Если из всех наборов повторений мы сохраним только последний, то в результате получится список предшествования в естественном для CLOS порядке. (Удаление всех повторений, кроме последнего, соответствует правилу 3 на стр. 192.) Такое поведение закреплено за встроенной в Common Lisp функцией `delete-duplicates`. Поэтому если мы передадим ей результат поиска в глубину, она вернет корректный список предшествования. После того как этот список создан, `rget` ищет в нем первый объект с заданным свойством.

Используя идею предшествования, мы можем сказать, например, что патриотичный негодяй является прежде всего негодяем и лишь потом патриотом:[1]

[1] То есть служит (serves) сначала себе (self) и лишь затем стране (country), что и продемонстрировано в данном примере. – *Прим. перев.*

```
> (setf scoundrel           (make-hash-table)
        patriot             (make-hash-table)
        patriotic-scoundrel (make-hash-table)
        (gethash 'serves scoundrel) 'self
        (gethash 'serves patriot)   'country
        (gethash :parents patriotic-scoundrel)
                 (list scoundrel patriot))
(#<Hash-Table C41C7E> #<Hash-Table C41F0E>)
> (rget 'serves patriotic-scoundrel)
SELF
T
```

На этом этапе мы имеем очень мощную, но уродливую и неэффективную программу. Пора приступать ко второму этапу – приведению наброска программы к виду, пригодному для использования.

17.3. Определение объектов

Во-первых, нам нужен нормальный способ создания объектов, скрывающий их истинное устройство. Если мы определим функцию для построения объектов, одним вызовом которой можно создать объект и определить его предков, то сможем строить список предшествования для объекта лишь однажды – при его создании, избегая тем самым повторения этой дорогостоящей операции каждый раз, когда нужно найти свойство или метод.

Если мы собираемся хранить список предшествования, вместо того чтобы каждый раз его воссоздавать, нужно учесть возможность его устаревания. Нашей стратегией будет хранение списка всех существующих объектов, а при модификации родителей какого-либо из них мы переопределим список предшествования всех объектов, на которые это влияет. Да, на перестройку списка будут затрачиваться некоторые ресурсы, но мы все равно останемся в выигрыше, так как обращение к свойствам – намного более частая операция, чем добавление новых предков. Кроме того, это нисколько не уменьшит гибкость нашей программы; мы просто переносим затраты с часто используемой операции на редко используемую.

На рис. 17.4. показан новый код.° Глобальная переменная `*objs*` является списком всех объектов. Функция `parents` получает предков заданного объекта; `(setf parents)` не только назначает новых предков, но и вызывает `make-precedence` для перестройки любого списка предшествования, который также мог поменяться. Списки, как и прежде, строятся с помощью `precedence`.

Теперь для создания объектов вместо `make-hash-table` пользователи могут вызывать `obj`, которая создает новый объект и определяет всех его предков за один шаг. Мы также переопределили `rget`, чтобы воспользоваться преимуществом хранимых списков предшествования.

```
(defvar *objs* nil)

(defun parents (obj) (gethash :parents obj))

(defun (setf parents) (val obj)
  (prog1 (setf (gethash :parents obj) val)
         (make-precedence obj)))

(defun make-precedence (obj)
  (setf (gethash ipreclist obj) (precedence obj))
  (dolist (x *objs*)
    (if (member obj (gethash :preclist x))
        (setf (gethash :preclist x) (precedence x)))))

(defun obj (&rest parents)
  (let ((obj (make-hash-table)))
    (push obj *objs*)
    (setf (parents obj) parents)
    obj))

(defun rget (prop obj)
  (dolist (c (gethash :preclist obj))
    (multiple-value-bind (val in) (gethash prop c)
      (if in (return (values val in))))))
```

Рис. 17.4. *Создание объектов*

17.4. Функциональный синтаксис

Усовершенствовать стоит и синтаксис передачи сообщений. Использование `tell` не только загромождает запись, но и лишает нас возможности читать программу в обычной для Лиспа префиксной нотации:

```
(tell (tell obj 'find-owner) 'find-owner)
```

Мы можем избавиться от `tell`, определяя имена свойств как функции. Определим для этого макрос `defprop` (рис. 17.5). Необязательный аргумент `meth?`, будучи истинным, сигнализирует, что свойство должно считаться методом. В противном случае оно будет считаться слотом, и значение, получаемое `rget`, будет просто возвращаться.

Теперь, когда мы определили имя свойства,

```
(defprop find-owner t)
```

можно сослаться на него через вызов функции, и наш код снова станет читаться как Лисп:

```
(find-owner (find-owner obj))
```

```
(defmacro defprop (name &optional meth?)
  `(progn
     (defun ,name (obj &rest args)
       ,(if meth?
            `(run-methods obj ',name args)
            `(rget ',name obj)))
     (defun (setf ,name) (val obj)
       (setf (gethash ',name obj) val))))

(defun run-methods (obj name args)
  (let ((meth (rget name obj)))
    (if meth
        (apply meth obj args)
        (error "No ~A method for "A." name obj))))
```

Рис. 17.5. *Функциональный синтаксис*

Теперь наш предыдущий пример станет более читаемым:

```
> (progn
    (setf scoundrel           (obj)
          patriot             (obj)
          patriotic-scoundrel (obj scoundrel patriot))
    (defprop serves)
    (setf (serves scoundrel) 'self
          (serves patriot)   'country)
    (serves patriotic-scoundrel))
SELF
T
```

17.5. Определение методов

До сих пор мы определяли методы следующим образом:

```
(defprop area t)

(setf circle-class (obj))

(setf (aref circle-class)
      #'(lambda (c) (* pi (expt (radius c) 2))))
```

Вызывая первый метод из `cdr` списка предшествования, мы можем получить внутри метода эффект встроенного оператора `call-next-method`. К примеру, если мы захотим печатать что-либо во время вычисления площади круга, мы скажем:

```
(setf grumpy-circle (obj circle-class))

(setf (area grumpy-circle)
      #'(lambda (c)
          (format t "How dare you stereotype me!~%")
```

```
            (funcall (some #'(lambda (x) (gethash 'area x))
                           (cdr (gethash :preclist c)))
                     c)))
```

Данный funcall эквивалентен call-next-method, но он открывает больше своего внутреннего устройства, чем следовало бы.

Макрос defmeth на рис. 17.6 предоставляет более удобный способ создания методов, а также облегчает вызов внутри них следующего метода.

```
(defmacro defmeth (name obj parms &rest body)
  (let ((gobj (gensym)))
    `(let ((,gobj ,obj))
       (setf (gethash ',name ,gobj)
             (labels ((next () (get-next ,gobj ',name)))
               #'(lambda ,parms ,@body))))))

(defun get-next (obj name)
  (some #'(lambda (x) (gethash name x))
        (cdr (gethash :preclist obj))))
```

Рис. 17.6. *Определение методов*

Вызов defmeth раскрывается в setf, внутри которого находится labels-выражение, задающее функцию next для получения следующего метода. Она работает наподобие next-method-p (стр. 182), но возвращает объект, который может быть вызван, то есть служит еще и как call-next-method.° Теперь предыдущие два метода можно определить следующим образом:

```
(defmeth area circle-class (c)
  (* pi (expt (radius c) 2)))

(defmeth area grumpy-circle (c)
  (format t "How dare you stereotype me!~%")
  (funcall (next) c))
```

Обратите внимание, каким образом в определении defmeth используется захват символов. Тело метода вставляется в контекст, в котором локально определена функция next.

17.6. Экземпляры

До сих пор мы не делали различий между классами и экземплярами, используя для них общий термин: *объект*. Конечно, такая унификация дает определенную гибкость и удобство, но она крайне неэффективна. В большинстве объектно-ориентированных приложений граф наследования тяготеет к низу. Например, в задаче симуляции трафика типы транспортных средств представляются менее чем десятью классами, а вот объектов, представляющих отдельные автомобили, сотни. Посколь-

ку у всех них будут одинаковые списки предшествования, создавать и хранить их по отдельности – пустая трата ресурсов времени и памяти.

На рис. 17.7 представлен макрос inst, производящий экземпляры. Экземпляры подобны объектам (которые мы теперь можем называть классами), но имеют лишь одного предка и не содержат списка предшествования. Кроме того, они не включаются в список *objs*. Теперь наш предыдущий пример может быть несколько изменен:

```
(setf grumpy-circle (inst circle-class))
```

Поскольку некоторые объекты отныне не содержат списков предшествования, мы переопределили rget и get-next, чтобы они могли сразу получать доступ к единственному предку. Прирост производительности никак не сказался на гибкости. Мы можем делать с экземплярами все, что могли делать с любыми объектами, в том числе создавать их экземпляры и переназначать предков. В последнем случае (setf parents) будет эффективно преобразовывать объект в «класс».

```
(defun inst (parent)
  (let ((obj (make-hash-table)))
    (setf (gethash :parents obj) parent)
  obj))

(defun rget (prop obj)
  (let ((prec (gethash :preclist obj)))
    (if prec
        (dolist (c prec)
          (multiple-value-bind (val in) (gethash prop c)
            (if in (return (values val in)))))
        (multiple-value-bind (val in) (gethash prop obj)
          (if in
              (values val in)
              (rget prop (gethash :parents obj)))))))

(defun get-next (obj name)
  (let ((prec (gethash :preclist obj)))
    (if prec
        (some #'(lambda (x) (gethash name x))
              (cdr prec))
        (get-next (gethash obj :parents) name))))
```

Рис. 17.7. *Создание экземпляров*

17.7. Новая реализация

Пока что ни одно из наших улучшений не сказалось отрицательно на гибкости программы. Обычно на последних этапах разработки Лисп-программа жертвует частью своей гибкости ради эффективности. И наша

программа не исключение. Представив все объекты в виде хеш-таблиц, мы не только получаем избыточную гибкость, но платим за это слишком высокую цену. В этом разделе мы перепишем нашу программу, чтобы представить объекты в виде простых векторов.

Да, мы отказываемся от возможности добавлять новые свойства на лету – до сих пор мы могли определить новое свойство любого объекта, просто сославшись на него. Но мы по-прежнему сможем делать это с помощью переопределения объектов. При создании класса нам потребуется передать ему список новых свойств, а при создании экземпляров они будут получать свойства через наследование.

В предыдущей реализации, по сути, не было разделения между классами и экземплярами. Экземпляром считался класс, имеющий лишь одного предка, и переназначение предков превращало экземпляр в класс. В новой реализации у нас будет полноценное деление между этими двумя типами объектов, но мы больше не сможем преобразовывать экземпляры в классы.

Код на рис. 17.8–17.10 полностью представляет новую реализацию. Рисунок 17.8 содержит новые операторы для определения классов и экземпляров. Классы и экземпляры представляются в виде векторов. Первые три элемента содержат информацию, необходимую самой программе, а первые три макроса на рис. 17.8 позволяют ссылаться на нее:

1. Поле `parents` аналогично записи `:parents` в ранее использовавшихся хеш-таблицах. Для класса оно будет содержать список родительских классов, для экземпляра – один родительский класс.
2. Поле `layout` (размещение) будет содержать вектор с именами свойств, который определяет их размещение в векторе самого класса или экземпляра, начиная с его четвертого элемента[1].
3. Поле `preclist` аналогично записи `:preclist` в ранее использовавшихся хеш-таблицах. Для класса оно будет содержать список предшествования, для экземпляра – `nil`.

Так как эти операторы – макросы, то они могут быть первыми аргументами `setf` (см. раздел 10.6).

Макрос `class` предназначен для создания классов. Он принимает необязательный список суперклассов, за которым следуют ноль или более имен свойств. Макрос возвращает объект, представляющий класс. В новом классе будут объединены его собственные свойства и свойства, унаследованные от суперклассов.

```
> (setf *print-array* nil
        geom-class   (class nil area)
        circle-class (class (geom-class) radius))
#<Simple-Vector T 5 C6205E>
```

[1] Первые 3 элемента одинаковы для всех объектов. Это родители, собственно `layout` и список предшествования. – *Прим. перев.*

```
(defmacro parents (v) `(svref ,v 0))
(defmacro layout (v) `(the simple-vector (svref ,v 1)))
(defmacro preclist (v) `(svref ,v 2))

(defmacro class (&optional parents &rest props)
  `(class-fn (list ,@parents) ',props))

(defun class-fn (parents props)
  (let* ((all (union (inherit-props parents) props))
         (obj (make-array (+ (length all) 3)
                          :initial-element :nil)))
    (setf (parents obj)  parents
          (layout obj)   (coerce all 'simple-vector)
          (preclist obj) (precedence obj))
    obj))

(defun inherit-props (classes)
  (delete-duplicates
    (mapcan #'(lambda (c)
                (nconc (coerce (layout c) 'list)
                       (inherit-props (parents c))))
            classes)))

(defun precedence (obj)
  (labels ((traverse (x)
             (cons x
                   (mapcan #'traverse (parents x)))))
    (delete-duplicates (traverse obj))))

(defun inst (parent)
  (let ((obj (copy-seq parent)))
    (setf (parents obj)  parent
          (preclist obj) nil)
    (fill obj :nil :start 3)
    obj))
```

Рис. 17.8. *Реализация с помощью векторов: создание объектов*

Здесь мы создали два класса: `geom-class` не имеет суперклассов и содержит только одно свойство `area`; `circle-class` является подклассом `geom-class` и имеет дополнительное свойство `radius`.[1] Функция `layout` позволяет увидеть имена последних двух из пяти его полей:[2]

```
> (coerce (layout circle-class) 'list)
(AREA RADIUS)
```

[1] При отображении классов `*print-array*` должен быть `nil`. Первый элемент в списке `preclist` любого класса сам является классом, поэтому попытка отображения такой структуры приведет к попаданию в бесконечный цикл.

[2] Вектор приводится к списку для просмотра его содержимого. Если `*print-array*` установлен в `nil`, содержимое вектора не отображается.

Макрос class – это просто интерфейс к функции class-fn, которая выполняет основную работу. Она вызывает inherit-props для сборки списка свойств всех предков нового объекта, создает вектор необходимой длины и устанавливает три первых его элемента. (preclist строится с помощью функции precedence, которую мы практически не изменили.) Остальные поля класса устанавливаются в :nil, указывая на то, что они не инициализированы. Чтобы получить свойство area класса circle-class, мы можем сказать:

```
> (svref circle-class
        (+ (position 'area (layout circle-class)) 3))
:NIL
```

Позже мы определим функции доступа, которые будут делать это автоматически.

Наконец, функция inst используется для создания экземпляров. Она не обязана быть макросом, так как принимает лишь один аргумент:

```
> (setf our-circle (inst circle-class))
#<Simple-Vector T 5 C6464E>
```

Полезно сравнить inst и class-fn, которые выполняют похожие задачи. Так как экземпляры имеют лишь одного родителя, то нет необходимости определять, какие свойства наследуются. Экземпляр может просто скопировать размещение родительского класса. Кроме того, нет необходимости строить список предшествования, поскольку у экземпляра его попросту нет. Таким образом, создание экземпляров намного быстрее создания классов. Но так и должно быть, ведь, как правило, новые экземпляры создаются намного чаще, чем новые классы.

Теперь, чтобы построить иерархию классов и экземпляров, нам потребуются функции чтения и записи свойств. Первой функцией на рис. 17.9 является новая версия rget. По форме она напоминает rget с рис. 17.7. Она имеет две ветки – для классов и экземпляров соответственно.

1. Если объект является классом, мы обходим его список предшествования до тех пор, пока не найдем объект, имеющий значение искомого свойства, не равное :nil. Если такой объект не найден, возвращаем :nil.
2. Если объект является экземпляром, то ищем лишь локально, среди его свойств, а если не находим, то рекурсивно вызываем rget.

У rget появился новый третий аргумент – next?, предназначение которого будет объяснено позже. Сейчас достаточно сказать, что если он nil, то rget ведет себя как обычно.

Функция lookup и обратная ей играют роль gethash в старой версии rget. Они ищут в объекте свойство с заданным именем или устанавливают новое. Этот вызов функции эквивалентен предыдущей версии с svref:

```
> (lookup 'area circle-class)
:NIL
```

```
(declaim (inline lookup (setf lookup)))

(defun rget (prop obj next?)
  (let ((prec (preclist obj)))
    (if prec
        (dolist (c (if next? (cdr prec) prec) :nil)
          (let ((val (lookup prop c)))
            (unless (eq val :nil) (return val))))
        (let ((val (lookup prop obj)))
          (if (eq val :nil)
              (rget prop (parents obj) nil)
              val)))))

(defun lookup (prop obj)
  (let ((off (position prop (layout obj) :test #'eq)))
    (if off (svref obj (+ off 3)) :nil)))

(defun (setf lookup) (val prop obj)
  (let ((off (position prop (layout obj) :test #'eq)))
    (if off
        (setf (svref obj (+ off 3)) val)
        (error "Can't set ~A of ~A." val obj))))
```

Рис. 17.9. *Реализация с помощью векторов: доступ*

Поскольку `setf` для `lookup` также определено, мы можем определить метод `area` для `circle-class`:

```
(setf (lookup 'area circle-class)
      #'(lambda (c)
          (* pi (expt (rget 'radius c nil) 2))))
```

В этой программе, как и в ее предыдущей версии, нет четкого разделения между классами и экземплярами, а «метод» – это просто функция в поле объекта. Но вскоре это будет скрыто за более удобным фасадом.

Рисунок 17.10 содержит оставшуюся часть нашей новой реализации.[1] Этот код не добавляет программе никаких новых возможностей, но делает ее более простой в использовании. Макрос `defprop`, по сути, остался тем же, просто теперь он вызывает `lookup` вместо `gethash`. Как и раньше, он позволяет ссылаться на свойства в функциональном стиле:

```
> (defprop radius)
(SETF RADIUS)
> (radius our-circle)
:NIL
```

[1] Данный листинг содержит исправление авторской опечатки, которая была обнаружена Тимом Мензисом (Tim Menzies) и опубликована в новостной группе comp.lang.lisp (19.12.2010). Эта опечатка не внесена автором в официальный список опечаток книги. – *Прим. перев.*

```
> (setf (radius our-circle) 2)
2
```

Если необязательный второй аргумент макроса defprop истинен, его вызов раскрывается в run-methods, который сохранился практически без изменений.

Наконец, функция defmeth предоставляет удобный способ определения методов. В этой версии есть три новых момента: она неявно вызывает defprop, использует lookup вместо gethash и вызывает rget вместо get-next (стр. 282) для получения следующего метода. Теперь мы видим причину добавления необязательного аргумента в rget: эта функция так похожа на get-next, что мы можем объединить их в одну за счет добавления дополнительного аргумента. Если этот аргумент истинен, rget выполняет роль get-next.

```
(declaim (inline run-methods))

(defmacro defprop (name &optional meth?)
  `(progn
     (defun ,name (obj &rest args)
       ,(if meth?
            `(run-methods obj ',name args)
            `(rget ',name obj nil)))
     (defun (setf ,name) (val obj)
       (setf (lookup ',name obj) val))))

(defun run-methods (obj name args)
  (let ((meth (rget name obj nil)))
    (if (not (eq meth :nil))
        (apply meth obj args)
        (error "No ~A method for ~A." name obj))))

(defmacro defmeth (name obj parms &rest body)
  (let ((gobj (gensym)))
    `(let ((,gobj ,obj))
       (defprop ,name t)
       (setf (lookup ',name ,gobj)
             (labels ((next () (rget ',name ,gobj t)))
               #'(lambda ,parms ,@body))))))
```

Рис. 17.10. *Реализация с помощью векторов: макросы интерфейса*

Теперь мы можем достичь того же эффекта, что и в предыдущем примере, но с помощью существенно более понятного кода:

```
(defmeth area circle-class (c)
  (* pi (expt (radius c) 2)))
```

Обратите внимание, что вместо вызова rget мы можем сразу вызвать radius, так как она была определена как функция при вызове defprop.

Благодаря неявному вызову `defprop` внутри `defmeth` мы можем аналогичным образом вызвать `area`, чтобы найти площадь `our-circle`:

```
> (area our-circle)
12.566370614359173
```

17.8. Анализ

Теперь у нас есть встроенный язык, приспособленный для написания реальных объектно-ориентированных приложений. Он простой, но для своего размера достаточно мощный. Кроме того, он будет достаточно быстрым для типичных приложений, в которых операции с экземплярами обычно выполняются намного чаще, чем с классами. Основной задачей усовершенствования кода было именно повышение производительности при работе с экземплярами.

В нашей программе создание новых классов происходит медленно и производит большое количество мусора. Но если классы не создаются в те моменты, когда критична скорость, это вполне приемлемо. Что действительно должно происходить быстро, так это доступ и создание экземпляров. В плане скорости доступа к экземплярам наша программа достигла предела, и дальнейшие усовершенствования могут быть связаны с оптимизацией при компиляции.° То же самое касается и создания экземпляров. И никакие операции с экземплярами не требуют выделения памяти, за исключением, разумеется, создания вектора, представляющего сам экземпляр. Сейчас такие векторы размещаются в памяти динамически, что вполне естественно, но при желании можно избежать динамического выделения памяти, применив стратегию, описанную в разделе 13.4.

Наш встроенный язык является характерным примером программирования на Лиспе. Уже тот факт, что он встроенный, говорит об этом. Но показательно и то, как Лисп позволил нам развивать программу от небольшой ограниченной версии к мощной, но неэффективной, а затем к эффективной, но слегка ограниченной.

Репутация Лиспа как медленного языка происходит не из его природы (еще с 1980-х годов компиляторы Лиспа способны генерировать не менее быстрый код, чем компиляторы C), а из того, что многие программисты останавливаются на втором этапе развития программы. Как писал Ричард Гэбриел:

> «В Лиспе очень легко написать программу с плохой производительностью; в C это практически невозможно».°

Да, это так, но это высказывание может пониматься как за, так и против Лиспа:

1. Приобретая гибкость ценой скорости, вы облегчаете процесс написания программ в Лиспе; в C вы не имеете такой возможности.
2. Если не оптимизировать ваш Лисп-код, очень легко получить медленную программу.

Какая из интерпретаций описывает вашу программу, зависит всецело от вас. Но в Лиспе у вас как минимум есть возможность пожертвовать скоростью выполнения ради экономии вашего времени на ранних этапах создания программы. Для чего наш пример точно *не* подходит, так это для демонстрации модели работы CLOS (за исключением, пожалуй, прояснения механизма работы `call-next-method`). Да и как можно говорить о сходстве слонопотамного CLOS и нашего 70-строчного комарика. Действительно, различия между этими двумя программами более показательны, чем сходства. Во-первых, мы увидели, насколько широким понятием может быть объектная ориентированность. Наша программа будет помощнее, чем некоторые другие реализации объектно-ориентированного программирования, но тем не менее она обладает лишь частью мощи CLOS.

Наша реализация отличается от CLOS тем, что методы являются методами *определенного* объекта. Такая концепция методов приравнивает их к функциям, поведение которых управляется их первым аргументом. Обобщенные функции CLOS могут управляться любыми из своих аргументов, а методами считаются компоненты обобщенной функции. Если они будут специфицированы только по первому аргументу, то можно создать иллюзию, что методы принадлежат определенным классам. Но представление о CLOS в терминах модели передачи сообщений лишь запутает вас, так как CLOS значительно превосходит ее.

Отметим один из недостатков CLOS: ее огромный размер и проработка скрывают то, насколько объектно-ориентированное программирование является лишь перефразировкой Лиспа. Пример, приведенный в данной главе, по крайней мере, проясняет этот момент. Если бы нас удовлетворяла старая модель передачи сообщений, нам бы хватило немногим более страницы кода для ее реализации. Объектно-ориентированное программирование – лишь одна из вещей, на которые способен Лисп. А теперь более интересный вопрос: на что еще он способен?

A

Отладка

В этом приложении показывается, как производить отладку в Лиспе, а также приводятся примеры некоторых часто встречающихся ошибок.

Циклы прерывания

Если вы просите Лисп сделать то, чего он не может, вычисление будет прервано сообщением об ошибке, и вы попадете в то, что называется *циклом прерывания* (*break loop*). Поведение этого цикла зависит от используемой реализации, но во всех случаях, как правило, отображаются следующие вещи: сообщение об ошибке, список опций и специальное приглашение.

Внутри цикла прерывания вы по-прежнему можете вычислять выражения так же, как и в toplevel. Находясь внутри этого цикла, вы можете выяснить причину ошибки и даже исправить ее на лету, после чего программа продолжит выполнение. Тем не менее наиболее частое действие, которое вы захотите произвести внутри цикла прерывания, – это выбраться из него. В большинстве ошибок, как правило, виноваты опечатки или мелкие оплошности, так что обычно вы просто прервете выполнение и вернетесь в toplevel. В гипотетической реализации для этого нужно набрать `:abort`:

```
> (/ 1 0)
Error: Division by zero.
       Options: :abort, :backtrace
>> :abort
>
```

Что вы будете вводить в своей реализации, зависит от нее.

Если вы находитесь в цикле прерывания и происходит еще одна ошибка, то вы попадаете в еще один цикл прерывания.[1] Большинство реализаций Лиспа показывает уровень цикла, на котором вы находитесь, с помощью нескольких приглашений или числа перед приглашением:

```
>> (/ 2 0)
Error: Division by zero.
       Options: :abort, :backtrace, :previous
>>>
```

Теперь глубина вложенности отладчиков равна двум, и мы можем вернуться в предыдущий отладчик или же сразу попасть в toplevel.

Трассировка и обратная трассировка

Если ваша программа делает не то, что вы ожидали, иногда стоит прежде всего выяснить, *что именно* она делает. Если вы введете (trace foo), то Лисп будет выводить сообщение каждый раз, когда foo вызывается и завершается, показывая при этом переданные ей аргументы или возвращенные ею значения. Вы можете отслеживать таким образом поведение любой заданной пользователем функции.

Трасса обычно выравнивается по глубине вложенности вызова. Для функции, выполняющей обход, например, для следующей функции, добавляющей единицу к каждому непустому элементу дерева,

```
(defun tree1+ (tr)
  (cond ((null tr) nil)
        ((atom tr) (1+ tr))
        (t (cons (tree1+ (car tr))
                 (tree1+ (cdr tr))))))
```

трасса будет отображать структуру, обход которой происходит:

```
> (trace treel+)
(treel+)
> (treel+ '((1 . 3) 5 . 7))
1 Enter TREE1+ ((1 . 3) 5 . 7)
  2 Enter TREE1+ ( 1 . 3 )
    3 Enter TREE1+ 1
    3 Exit TREE1+ 2
    3 Enter TREE1+ 3
    3 Exit TREE1+ 4
  2 Exit TREE1+ (2 . 4)
  2 Enter TREE1+ (5 . 7)
    3 Enter TREE1+ 5
    3 Exit TREE1+ 6
    3 Enter TREE1+ 7
    3 Exit TREE1+ 8
```

[1] Как раз поэтому отладчик в Лиспе называют break loop. – *Прим. перев.*

```
  2 Exit TREE1+ (6 . 8)
1 Exit TREE1+ ((2 . 4) 6 . 8)
((2 . 4) 6 . 8)
```

Для отключения трассировки `foo` введите `(untrace foo)`; для отключения трассировки всех функций введите `(untrace)`.

Более гибкой альтернативой трассировки является добавление в код диагностической печати. Пожалуй, этот метод, ставший классическим, применяется в десять раз чаще, чем изощренные отладочные утилиты. Это еще одна причина, по которой полезно интерактивное переопределение функций.

Обратная трасса (*backtrace*) – это список всех вызовов, находящихся на стеке, который создается из цикла прерывания после того, как произошла ошибка. Если при трассировке мы как бы говорим: «Покажи, что ты делаешь», то при обратной трассировке мы задаем вопрос: «Как мы сюда попали?». Эти две операции дополняют друг друга. Трассировка покажет каждый вызов заданных функций во всем дереве вызовов, а обратная трассировка – вызовы всех функций в выбранной части кода (на пути от toplevel к месту возникновения ошибки).

В типичной реализации мы получим обратную трассу, введя `:backtrace` в отладчике:

```
> (tree1+ '((1 . 3) 5 . A))
Error: A is not a valid argument to 1+.
       Options: :abort, :backtrace
>> :backtrace
(1+ A)
(TREE1+ A)
(TREE1+ (5 . A))
(TREE1+ ((1 . 3) 5 . A))
```

С помощью обратной трассировки поймать ошибку иногда довольно легко. Для этого достаточно посмотреть на цепочку вызовов. Еще одним преимуществом функционального программирования (раздел 2.12) является то, что все баги проявятся при обратной трассировке. В полностью функциональном коде все, что могло повлиять на возникновение ошибки, при ее появлении находится на стеке вызовов функции.

Количество информации, содержащееся в обратной трассе, зависит от используемой реализации. Одна реализация может выводить полную историю всех незавершенных вызовов с их аргументами, а другая может не выводить практически ничего. Имейте в виду, что обычно трассировка (и обратная трассировка) более информативна для интерпретируемого кода, нежели для скомпилированного. Это еще одна причина отложить компиляцию вашей программы до тех пор, пока вы не будете полностью уверены, что она работает. Исторически отладку вели в режиме интерпретации и лишь затем компилировали отлаженную версию. Однако взгляды на этот вопрос постепенно меняются: по меньшей мере две реализации Common Lisp вовсе не содержат интерпретатор.

Когда ничего не происходит

Не все баги приводят к прерыванию вычислений. Даже более неприятной является ситуация, когда Лисп полностью игнорирует все ваши действия. Обычно это означает, что вы попали в бесконечный цикл. Если это происходит в процессе итерации, то Лисп со счастливой улыбкой будет выполнять ее бесконечно. Если же вы столкнулись с бесконечной рекурсией (которая не оптимизирована как хвостовая), то вы рано или поздно получите сообщение, информирующее об отсутствии свободного места на стеке:

```
> (defun blow-stack () (1+ (blow-stack)))
BLOW-STACK
> (blow-stack)
Error: Stack overflow.
```

Если же этого не происходит, то вам остается лишь принудительно прервать вычисления, а затем выйти из цикла прерывания.

Иногда программа, решающая трудоемкую задачу, исчерпывает место на стеке не по причине бесконечной рекурсии. Но это происходит довольно редко, и исчерпание стека, как правило, свидетельствует о программной ошибке.

При создании рекурсивных функций и словесном описании рекурсивного алгоритма часто забывают базовый случай. Например, мы можем сказать, что `obj` принадлежит списку `lst`, если он является его первым элементом либо принадлежит к остатку `lst`. Строго говоря, это не так. Мы должны добавить еще одно условие: `obj` не принадлежит `lst`, если `lst` пуст. В противном случае наше описание будет соответствовать бесконечной рекурсии.

В Common Lisp `car` и `cdr` возвращают `nil`, если их аргументом является `nil`:

```
> (car nil)
NIL
> (cdr nil)
NIL
```

Поэтому если мы пропустим базовое условие в определении `member`,

```
(defun our-member (obj lst)                    ; wrong
  (if (eql (car lst) obj)
      lst
      (our-member obj (cdr lst))))
```

то получим бесконечную рекурсию, если искомый объект будет отсутствовать в списке. Когда мы достигнем конца списка, так и не найдя элемент, рекурсивный вызов будет иметь следующий вид:

```
(our-member obj nil)
```

В корректном определении (стр. 33) базовое условие приведет к остановке рекурсии, возвращая nil. В нашем ошибочном случае функция будет покорно искать car и cdr от nil, которые также равны nil, поэтому процесс будет повторяться снова и снова.

Если причина бесконечного процесса неочевидна, как в нашем случае, вам поможет трассировка. Бесконечные циклы бывают двух видов. Вам повезло, если источником бесконечного повторения является структура программы. Например, в случае our-member трассировка тут же покажет, где проблема.

Более сложным случаем являются бесконечные циклы, причина которых – некорректно сформированные структуры данных. Например, попытка обхода циклического списка (стр. 215) приводит к бесконечному циклу. Подобные проблемы трудно обнаружить, так как они проявляют себя не сразу и возникают уже в коде, который не является их непосредственной причиной. Правильным решением является предупреждение подобных проблем (стр. 205): избегайте деструктивных операторов до тех пор, пока не убедитесь в корректности работы программы.

Если Лисп игнорирует ваши действия, причиной этого может быть ожидание ввода. Во многих системах нажатие Enter не приводит к каким-либо эффектам, если ввод выражения не завершен. Преимущество такого подхода заключается в том, что это дает возможность введения многострочных выражений. Но есть и недостаток: если вы забудете ввести закрывающую скобку или кавычку, Лисп будет ожидать завершения ввода выражения, в то время как вы будете думать, что ввели его полностью:

```
> (format t "for example ~A~% 'this)
```

Здесь мы пропустили закрывающую кавычку в конце строки форматирования. Нажатие на Enter не приведет к чему-либо, так как Лисп думает, что мы продолжаем набирать строку.

В некоторых реализациях вы можете вернуться на предыдущую строку и добавить завершающую кавычку. В тех системах, где возможность редактирования предыдущих строк не предусмотрена, универсальным решением является прерывание программы с последующим выходом из цикла прерывания.

Переменные без значения/ несвязанные переменные

Частой причиной жалоб со стороны Лиспа являются символы без значения или же несвязанные символы. Они могут вызвать несколько различных проблем.

Локальные переменные, подобные устанавливаемым в let или defun, действительны только внутри тела выражения, в котором они были соз-

даны. Попытка сослаться на такие переменные откуда-либо извне приведет к ошибке:

```
> (progn
    (let ((x 10))
      (format t "Here x = ~A.~%" x))
    (format t "But now it's gone...~%")
    x)
Here x = 10.
But now it's gone...
Error: X has no value.
```

Когда Лисп жалуется подобным образом, это значит, что вы случайно сослались на переменную, которая не существует. Поскольку в текущем окружении не существует локальной переменной `x`, Лисп думает, что мы ссылаемся на глобальную переменную или константу с тем же именем. Когда он не находит никакого значения, возникает ошибка. К такому же результату приведут опечатки в именах переменных.

Похожая проблема возникает, если мы пытаемся сослаться на функцию как на переменную:

```
> defun foo (x) (+ x 1))
Error: DEFUN has no value.
```

Поначалу это может озадачить: как это `defun` не имеет значения? Просто мы забыли открывающую скобку, и поэтому Лисп воспринимает символ `defun` (а это все, что он пока что считал) как глобальную переменную.

Иногда программисты забывают инициализировать глобальные переменные. Если вы не передаете второй аргумент `defvar`, глобальная переменная будет определена, но не инициализирована, и это может быть корнем проблемы.

Неожиданный nil

Если функции жалуются, что получили `nil` в качестве аргумента, то это признак того, что на ранних этапах работы в программе что-то пошло не так. Некоторые встроенные операторы возвращают `nil` при неудачном завершении. Но, поскольку `nil` – полноценный объект, проблема может обнаружить себя существенно позже, когда какая-нибудь другая часть вашей программы попытается воспользоваться этим якобы возвращаемым значением.

Например, представим, что наша функция, возвращающая количество дней в месяце, содержит баг – мы забыли про октябрь:

```
(defun month-length (mon)
  (case mon
    ((jan mar may jul aug dec) 31)
    ((apr jun sept nov) 30)
    (feb (if (leap-year) 29 28))))
```

Кроме того, мы имеем функцию для вычисления количества недель в месяце:

```
(defun month-weeks (mon) (/ (month-length mon) 7.0))
```

В таком случае при вызове month-weeks может произойти следующее:

```
> (month-weeks 'oct)
Error: NIL is not a valid argument to /.
```

Ни одно из заданных условий в case-выражении не было выполнено, и case вернул nil. Затем month-weeks, которая ожидает число, передает это значение функции /, но та получает nil, что и приводит к ошибке.

В нашем примере источник и место проявления бага находятся рядом. Чем дальше они друг от друга, тем сложнее найти подобные баги. Для предотвращения подобных ситуаций в некоторых диалектах Лиспа прохождение cond и case без выполнения одного из вариантов вызывает ошибку. В Common Lisp в этой ситуации следует использовать ecase (см. раздел 14.6).

Переименование

Особенно зловредный баг появляется при переименовании функции или переменной в некоторых, но не во всех местах использования. Например, предположим, что мы определили следующую (неэффективную) реализацию функции вычисления глубины вложенного списка:

```
(defun depth (x)
  (if (atom x)
      1
      (1+ (apply #'max (mapcar #'depth x)))))
```

При первом же тестировании мы обнаружим, что она дает значение, завышенное на 1:

```
> (depth '((a)))
3
```

Исходное значение должно быть равно 0, а не 1. Исправим этот недочет, а заодно дадим функции более конкретное имя:

```
(defun nesting-depth (x)
  (if (atom x)
      0
      (1+ (apply #'max (mapcar #'depth x)))))
```

Теперь снова протестируем:

```
> (nesting-depth '((a)))
3
```

Мы получили тот же результат. В чем же дело? Да, ошибку мы исправили, но этот результат получается не из исправленного кода. Пригляди-

тесь внимательно, ведь мы забыли поменять имя в рекурсивном вызове, и наша новая функция по-прежнему вызывает некорректную функцию depth.

Путаница аргументов по ключу и необязательных аргументов

Если функция принимает одновременно необязательные аргументы и аргументы по ключу, частой ошибкой является случайная передача ключевого слова (по невнимательности) в качестве значения необязательного аргумента. Например, функция read-from-string имеет следующий список аргументов:

```
(read-from-string string &optional eof-error eof-value
                         &key start end preserve-whitespace)
```

Чтобы передать такой функции аргументы по ключу, нужно сначала передать ей все необязательные аргументы. Если мы забудем об этом, как в данном примере,

```
> (read-from-string "abcd" :start 2)
ABCD
4
```

то :start и 2 будут расценены как необязательные аргументы. Корректный вызов будет выглядеть следующим образом:

```
> (read-from-string "abcd" nil nil :start 2)
CD
4
```

Некорректные декларации

В главе 13 объяснялось, как определять декларации переменных и структур данных. Декларируя что-либо, мы даем обещание в дальнейшем следовать этой декларации, и компилятор полностью полагается на это допущение. Например, оба аргумента следующей функции обозначены как double-float,

```
(defun df* (a b)
  (declare (double-float a b))
  (* a b))
```

и поэтому компилятор сгенерировал код, предназначенный для умножения только чисел double-float.

Если df* вызывается с аргументами, не соответствующими задекларированному типу, то это вызовет ошибку или вовсе вернет мусор. А в одной из реализаций при передаче в эту функцию двух фиксированных целых происходит аппаратное прерывание:

```
> (df* 2 3)
Error: Interrupt.
```

Если у вас возникает подобная серьезная ошибка, с большой вероятностью ее причиной является неверная декларация типа.

Предупреждения

Иногда Лисп может жаловаться на что-либо без прерывания вычислений. Многие из этих предупреждений являются ложными тревогами. Например, часто возникают предупреждения, которыми сопровождается компиляция функций, содержащих незадекларированные или неиспользуемые переменные. Например, во втором вызове `map-int` (стр. 117) переменная *x* не используется. Если вы не хотите, чтобы компилятор каждый раз сообщал вам об этом факте, используйте декларацию `ignore`:

```
(map-int #'(lambda (x)
             (declare (ignore x))
             (random 100))
         10)
```

B

Лисп на Лиспе

Это приложение содержит определения 58 наиболее используемых операторов Common Lisp. Так как большая часть Лиспа написана (или может быть написана) на нем самом и при этом Лисп-программы невелики (или могут быть такими), это удобный способ объяснения самого языка.

Кроме того, данное упражнение демонстрирует, что концептуально Common Lisp не такой уж и большой язык, как может показаться. Большинство операторов Common Lisp являются, по сути, библиотечными функциями. Набор операторов, необходимых для определения остальных, довольно мал. Для определения тех операторов, которые мы рассмотрим в этой главе, достаточно наличия лишь следующих операторов:

```
apply aref backquote block car cdr ceiling char= cons defmacro
documentation eq error expt fdefinition function floor gensym
get-setf-expansion if imagpart labels length multiple-value-bind
nth-value quote realpart symbol-function tagbody type-of typep = + - / < >
```

Код, представленный здесь, написан таким образом, чтобы объяснять Common Lisp, но не реализовывать его. В полноценных реализациях эти операторы могут быть более производительными и лучше проводить проверку на наличие ошибок. Операторы определены в алфавитном порядке для упрощения их поиска. Если вы действительно хотите использовать эту реализацию Лиспа, определения макросов должны находиться перед любым кодом, который их использует.

```
(defun -abs (n)
  (if (typep n 'complex)
      (sqrt (+ (expt (realpart n) 2) (expt (imagpart n) 2)))
      (if (< n 0) (- n) n)))

(defun -adjoin (obj lst &rest args)
  (if (apply #'member obj lst args) lst (cons obj lst)))
```

```
(defmacro -and (&rest args)
  (cond ((null args) t)
        ((cdr args)  `(if ,(car args) (-and ,@(cdr args))))
        (t           (car args))))

(defun -append (&optional first &rest rest)
  (if (null rest)
      first
      (nconc (copy-list first) (apply #'-append rest))))

(defun -atom (x) (not (consp x)))

(defun -butlast (lst &optional (n 1))
  (nreverse (nthcdr n (reverse lst))))

(defun -cadr (x) (car (cdr x)))

(defmacro -case (arg &rest clauses)
  (let ((g (gensym)))
    `(let ((,g ,arg))
       (cond ,@(mapcar #'(lambda (cl)
                           (let ((k (car cl)))
                             `(,(cond ((member k '(t otherwise))
                                       t)
                                      ((consp k)
                                       `(member ,g ',k))
                                      (t `(eql ,g ',k)))
                               (progn ,@(cdr cl)))))
                       clauses)))))

(defun -cddr (x) (cdr (cdr x)))

(defun -complement (fn)
  #'(lambda (&rest args) (not (apply fn args))))

(defmacro -cond (&rest args)
  (if (null args)
      nil
      (let ((clause (car args)))
        (if (cdr clause)
            `(if ,(car clause)
                 (progn ,@(cdr clause))
                 (-cond ,@(cdr args)))
            `(or ,(car clause)
                 (-cond ,@(cdr args)))))))

(defun -consp (x) (typep x 'cons))

(defun -constantly (x) #'(lambda (&rest args) x))
```

```
(defun -copy-list (lst)
  (labels ((cl (x)
             (if (atom x)
                 x
                 (cons (car x)
                       (cl (cdr x))))))
    (cons (car lst)
          (cl (cdr lst)))))

(defun -copy-tree (tr)
  (if (atom tr)
      tr
      (cons (-copy-tree (car tr))
            (-copy-tree (cdr tr)))))

(defmacro -defun (name parms &rest body)
  (multiple-value-bind (dec doc bod) (analyze-body body)
    `(progn
       (setf (fdefinition ',name)
             #'(lambda ,parms
                 ,@dec
                 (block ,(if (atom name) name (second name))
                   ,@bod))
             (documentation ',name 'function)
             ,doc)
       ',name)))

(defun analyze-body (body &optional dec doc)
  (let ((expr (car body)))
    (cond ((and (consp expr) (eq (car expr) 'declare))
           (analyze-body (cdr body) (cons expr dec) doc))
          ((and (stringp expr) (not doc) (cdr body))
           (if dec
               (values dec expr (cdr body))
               (analyze-body (cdr body) dec expr)))
          (t (values dec doc body)))))

; Это определение, строго говоря, не является корректным; см let.

(defmacro -do (binds (test &rest result) &rest body)
  (let ((fn (gensym)))
    `(block nil
       (labels ((,fn ,(mapcar #'car binds)
                    (cond (,test ,@result)
                          (t (tagbody ,@body)
                             (,fn ,@(mapcar #'third binds))))))
         (,fn ,@(mapcar #'second binds))))))
```

```
(defmacro -dolist ((var lst &optional result) &rest body)
  (let ((g (gensym)))
    `(do ((,g ,lst (cdr ,g)))
         ((atom ,g) (let ((,var nil)) ,result))
       (let ((,var (car ,g)))
         ,@body))))

(defun -eql (x y)
  (typecase x
    (character (and (typep y 'character) (char= x y)))
    (number    (and (eq (type-of x) (type-of y))
                    (= x y)))
    (t         (eq x y))))

(defun -evenp (x)
  (typecase x
    (integer (= 0 (mod x 2)))
    (t       (error "non-integer argument"))))

(defun -funcall (fn &rest args) (apply fn args))

(defun -identity (x) x)

; Это определение не является полностью корректным: выражение
; (let ((&key 1) (&optional 2))) корректно, а вот его раскрытие - нет.

(defmacro -let (parms &rest body)
  `((lambda ,(mapcar #'(lambda (x)
                         (if (atom x) x (car x)))
                     parms)
      ,@body)
    ,@(mapcar #'(lambda (x)
                  (if (atom x) nil (cadr x)))
              parms)))

(defun -list (&rest elts) (copy-list elts))

(defun -listp (x) (or (consp x) (null x)))

(defun -mapcan (fn &rest lsts)
  (apply #'nconc (apply #'mapcar fn lsts)))

(defun -mapcar (fn &rest lsts)
  (cond ((member nil lsts) nil)
        ((null (cdr lsts))
         (let ((lst (car lsts)))
           (cons (funcall fn (car lst))
                 (-mapcar fn (cdr lst)))))
        (t
         (cons (apply fn (-mapcar #'car lsts))
               (apply #'-mapcar fn
                      (-mapcar #'cdr lsts))))))
```

```
(defun -member (x lst &key test test-not key)
  (let ((fn (or test
                (if test-not
                    (complement test-not))
                    #'eql)))
    (member-if #'(lambda (y)
                   (funcall fn x y))
               lst
               :key key)))

(defun -member-if (fn lst &key (key #'identity))
  (cond ((atom lst) nil)
        ((funcall fn (funcall key (car lst))) lst)
        (t (-member-if fn (cdr lst) :key key))))

(defun -mod (n m)
  (nth-value 1 (floor n m)))

(defun -nconc (&optional lst &rest rest)
  (if rest
      (let ((rest-conc (apply #'-nconc rest)))
        (if (consp lst)
            (progn (setf (cdr (last lst)) rest-conc)
                   lst)
            rest-conc))
      lst))

(defun -not (x) (eq x nil))

(defun -nreverse (seq)
  (labels ((nrl (lst)
             (let ((prev nil))
               (do ()
                   ((null lst) prev)
                 (psetf (cdr lst) prev
                        prev      lst
                        lst       (cdr lst)))))
           (nrv (vec)
             (let* ((len (length vec))
                    (ilimit (truncate (/ len 2))))
               (do ((i 0 (1+ i))
                    (j (1- len) (1- j)))
                   ((>= i ilimit) vec)
                 (rotatef (aref vec i) (aref vec j))))))
    (if (typep seq 'vector)
        (nrv seq)
        (nrl seq))))

(defun -null (x) (eq x nil))
```

```
(defmacro -or (&optional first &rest rest)
  (if (null rest)
      first
      (let ((g (gensym)))
        `(let ((,g ,first))
           (if ,g
               ,g
               (-or ,@rest))))))

; Эта функция не входит в Common Lisp, но требуется здесь
; для нескольких других определений.

(defun pair (lst)
  (if (null lst)
      nil
      (cons (cons (car lst) (cadr lst))
            (pair (cddr lst)))))

(defun -pairlis (keys vals &optional alist)
  (unless (= (length keys) (length vals))
    (error "mismatched lengths"))
  (nconc (mapcar #'cons keys vals) alist))

(defmacro -pop (place)
  (multiple-value-bind (vars forms var set access)
                       (get-setf-expansion place)
    (let ((g (gensym)))
      `(let* (,@(mapcar #'list vars forms)
              (,g ,access)
              (,(car var) (cdr ,g)))
         (prog1 (car ,g)
           ,set)))))

(defmacro -prog1 (arg1 &rest args)
  (let ((g (gensym)))
    `(let ((,g ,arg1))
       ,@args
       ,g)))

(defmacro -prog2 (arg1 arg2 &rest args)
  (let ((g (gensym)))
    `(let ((,g (progn ,arg1 ,arg2)))
       ,@args
       ,g)))

(defmacro -progn (&rest args) `(let nil ,@args))
```

```
(defmacro -psetf (&rest args)
  (unless (evenp (length args))
    (error "odd number of arguments"))
  (let* ((pairs (pair args))
         (syms (mapcar #'(lambda (x) (gensym))
                       pairs)))
    `(let ,(mapcar #'list
                   syms
                   (mapcar #'cdr pairs))
       (setf ,@(mapcan #'list
                       (mapcar #'car pairs)
                       syms)))))

(defmacro -push (obj place)
  (multiple-value-bind (vars forms var set access)
                       (get-setf-expansion place)
    (let ((g (gensym)))
      `(let* ((,g ,obj)
              ,@(mapcar #'list vars forms)
              (,(car var) (cons ,g ,access)))
         ,set))))

(defun -rem (n m)
  (nth-value 1 (truncate n m)))

(defmacro -rotatef (&rest args)
  `(psetf ,@(mapcan #'list
                    args
                    (append (cdr args)
                            (list (car args))))))

(defun -second (x) (cadr x))

(defmacro -setf (&rest args)
  (if (null args)
      nil
      `(setf2 ,@args)))

(defmacro setf2 (place val &rest args)
  (multiple-value-bind (vars forms var set)
                       (get-setf-expansion place)
    `(progn
       (let* (,@(mapcar #'list vars forms)
              (,(car var) ,val))
         ,set)
       ,@(if args `((setf2 ,@args)) nil))))

(defun -signum (n)
  (if (zerop n) 0 (/ n (abs n))))
```

```
(defun -stringp (x) (typep x 'string))

(defun -tailp (x y)
  (or (eql x y)
      (and (consp y) (-tailp x (cdr y)))))

(defun -third (x) (car (cdr (cdr x))))

(defun -truncate (n &optional (d 1))
  (if (> n 0) (floor n d) (ceiling n d)))

(defmacro -typecase (arg &rest clauses)
  (let ((g (gensym)))
    `(let ((,g ,arg))
       (cond ,@(mapcar #'(lambda (cl)
                           `((typep ,g ',(car cl))
                             (progn ,@(cdr cl))))
                       clauses)))))

(defmacro -unless (arg &rest body)
  `(if (not ,arg)
       (progn ,@body)))

(defmacro -when (arg &rest body)
  `(if ,arg (progn ,@body)))

(defun -1+ (x) (+ x 1))

(defun -1- (x) (- x 1))

(defun ->= (first &rest rest)
  (or (null rest)
      (and (or (> first (car rest)) (= first (car rest)))
           (apply #'->= rest))))
```

C

Изменения в Common Lisp

ANSI Common Lisp существенно отличается от Common Lisp, описанного в 1984 году в первом издании «Common Lisp: the Language» Гая Стила. Он также отличается (хотя и в меньшей степени) от описания языка во втором издании 1990 года. Это приложение описывает наиболее существенные различия. Изменения, внесенные с 1990 года, перечислены отдельно в последнем разделе.

Основные дополнения

1. Объектная система Common Lisp (CLOS) стала частью языка.
2. Макрос `loop` теперь реализует встроенный язык с инфиксным синтаксисом.
3. Common Lisp теперь включает набор новых операторов, которые в общем называются системой особых условий и предназначены для сигнализации и обработки ошибок и других условий.
4. Common Lisp теперь предоставляет специальную поддержку и управление процессом красивой печати (pretty printing).

Отдельные дополнения

1. Добавлены следующие операторы:

```
complement                  nth-value
declaim                     print-unreadable-object
defpackage                  readtable-case
delete-package              row-major-aref
destructuring-bind          stream-external-format
fdefinition                 with-compilation-unit
file-string-length          with-hash-table-iterator
```

```
function-lambda-expression  with-package-iterator
load-time-value             with-standard-io-syntax
map-into
```

2. Добавлены следующие глобальные переменные:

```
*debugger-hook*    *read-eval*    *print-readably*
```

Функции

1. Идея имени функции была обобщена и теперь включает выражения вида (setf *f*). Подобные выражения теперь принимаются любыми операторами или декларациями, которые ожидают имя функции. Новая функция fdefinition действует подобно symbol-function, но принимает имя функции в новом более общем смысле.
2. Тип function больше не включает fboundp-символы и лямбда-выражения. Символы (но не лямбда-выражения) по-прежнему могут использоваться там, где ожидается функциональный объект. Лямбда-выражения могут быть преобразованы в функции с помощью coerce.
3. Символы, используемые как имена аргументов по ключу, больше не обязаны быть ключевыми словами. (Обратите внимание, что символы не из пакета keyword при таком использовании должны выделяться кавычками.)
4. Для остаточных аргументов (&rest) не гарантируется новое выделение памяти, поэтому изменять их деструктивно небезопасно.
5. Локальные функции, определенные через flet или labels, а также результаты раскрытия defmacro, macrolet или defsetf неявно заключены в блок block, имя которого соответствует имени определяемого объекта.
6. Функция, имеющая интерпретируемое определение в ненулевом лексическом окружении (то есть определенная в toplevel через defun внутри let) не может быть скомпилирована.

Макросы

1. Были введены макросы компиляции и символы-макросы, а также соответствующие операторы.
2. Теперь раскрытие макроса определяется в том окружении, где был совершен вызов defmacro. По этой причине в ANSI Common Lisp код раскрытия макроса может ссылаться на локальные переменные:

```
(let ((op 'car))
  (defmacro pseudocar (lst)
    `(,op ,lst)))
```

В 1984 году раскрытие макроса определялось в нулевом окружении (то есть в toplevel).

3. Гарантируется, что макровызовы не будут перераскрываться в скомпилированном коде.
4. Макровызовы теперь могут быть точечными списками.
5. Макросы теперь не могут раскрываться как `declare`.

Вычисление и компиляция

1. Переопределен специальный оператор `eval-when`, все его оригинальные ключи объявлены нежелательными.
2. Функция `compile-file` теперь принимает аргументы `:print` и `:verbose`. Их значения по умолчанию хранятся в новых глобальных переменных `*compile-print*` и `*compile-verbose*`.
3. Новые переменные `*compile-file-pathname*` и `*compile-file-truename*` связываются на время вызова `compile-file`, как и `*load-pathname*` и `*load-truename*` в процессе `load`.
4. Добавлена декларация `dynamic-extent`.
5. Добавлен параметр компиляции `debug`.
6. Удален специальный оператор `compiler-let`.
7. Удален макрос чтения `#,`.
8. Удалена глобальная переменная `*break-on-warnings*`; на замену ей введена более общая `*break-on-signals*`.

Побочные эффекты

1. Метод `setf` теперь может выполнять несколько присваиваний.
2. Более четко определены способы изменения аргументов многими деструктивными функциями. Например, большинство операторов, которые могут модифицировать списки, не только имеют разрешение на это, но и обязаны это делать. Теперь подобные деструктивные функции *могут* применяться ради получения побочных эффектов, а не возвращаемых значений.
3. Теперь явным образом запрещено использование отображающих функций (а также макросов типа `dolist`) для изменения последовательностей, проход по которым они выполняют.

Символы

1. Новая глобальная переменная `*gensym-counter*` содержит целое число, используемое для получения печатных имен символов, создаваемых `gensym`. В 1984 году этот счетчик мог быть неявно сброшен передачей целого числа аргументом `gensym`; теперь это действие признано нежелательным.

2. Модификация строки, используемой как имя символа, теперь считается ошибкой.
3. Функция `documentation` стала обобщенной функцией.

Списки

1. Функции `assoc-if`, `assoc-if-not`, `rassoc-if`, `rassoc-if-not` и `reduce` теперь принимают аргумент `:key`.
2. Функция `last` теперь имеет второй необязательный параметр, задающий длину возвращаемого хвоста.
3. С добавлением `complement` использование функций `-if-not` и аргумента по ключу `:test-not` признано нежелательным.

Массивы

1. Добавлены новые функции, позволяющие программам запрашивать информацию об обновлении типов (type-upgrading) элементов массивов и комплексных чисел.
2. Индексы массивов теперь определены как `fixnum`.

Строки и знаки

1. Удален тип `string-char`. Тип `string` больше не идентичен типу (`vector string-char`). Тип `character` разделен на два новых подтипа: `base-char` и `extended-char`. Тип `string` получил новый подтип `base-string`, который, в свою очередь, имеет новый подтип `simple-base-string`.
2. Некоторые функции для создания строк теперь могут принимать аргумент `:element-type`.
3. Упразднены атрибуты знаков `font` и `bits`, а также все связанные с ними функции и константы. Единственным атрибутом знака, определенным в Common Lisp, теперь является `code`.
4. Большинство строковых функций могут преобразовывать аргументы одинаковых типов в строки. Так, (`string= 'x 'x`) теперь вернет `t`.

Структуры

1. Отныне не обязательно задавать какие-либо слоты в теле вызова `defstruct`.
2. Последствия переопределения структуры, то есть двукратного вызова `defstruct` с одним именем, не определены.

Хеш-таблицы

1. Функция `equalp` теперь может использоваться в хеш-таблицах.
2. Добавлены новые функции доступа, позволяющие программам обращаться к свойствам хеш-таблиц: `hash-table-rehash-size`, `hash-table-rehash-threshold`, `hash-table-size` и `hash-table-test`.

Ввод-вывод

1. Введена концепция логических путей, а также соответствующие операторы.
2. Введены несколько новых типов потоков, а также соответствующие предикаты и функции доступа.
3. Введены директивы форматирования `~_`, `~W` и `~I`. Директивы `~D`, `~B`, `~O`, `~X` и `~R` теперь принимают дополнительный аргумент.
4. Функции `write` и `write-to-string` принимают пять новых аргументов по ключу.
5. Для ввода путей добавлен новый макрос чтения `#P`.
6. Новая переменная `*print-readably*` может использоваться для принудительного вывода в читаемом виде.

Числа

1. Теперь тип `fixnum` имеет размер не менее 16 бит.
2. Добавлены восемь новых констант, ограничивающих допустимые значения нормализованных чисел с плавающей запятой.
3. Добавлен тип `real`; он является подтипом для `number` и надтипом для `rational` и `float`.

Пакеты

1. Изменено соглашение о способе определения пакетов в файлах с исходными кодами. Вызовы функций, связанных с пакетами, из top-level больше не обрабатываются компилятором. Вместо этого введен рекомендованный к использованию макрос `defpackage`. Старая функция `in-package` заменена на макрос с тем же именем. Новый макрос не вычисляет свой аргумент, не принимает аргумент `:use` и не создает пакеты неявно.
2. Пакеты `lisp` и `user` переименованы в `common-lisp` и `common-lisp-user`. Вновь создаваемые пакеты не используют `common-lisp` по умолчанию, как это происходило с пакетом `lisp`.
3. Имена встроенных переменных, функций, макросов и специальных операторов принадлежат `common-lisp`. Попытка переопределения, свя-

зывания, трассировки и деклараций применительно к любым встроенным операторам считается ошибкой.

Типы

1. Добавлен спецификатор типов `eql`.
2. Удален тип `common` и функция `commonp`.

Изменения с 1990 года

1. Добавлены следующие операторы:

```
allocate-instance          ensure-directories-exist
array-displacement         lambda
constantly                 read-sequence
define-symbol-macro        write-sequence
```

2. Удалены следующие операторы и переменные:

```
applyhook                  function-information
*applyhook*                get-setf-method
augment-environment        generic-flet
declare                    generic-function
enclose                    generic-labels
evalhook                   parse-macro
*evalhook*                 variable-information
declaration-information    with-added-methods
define-declaration
```

   Вместо `get-setf-method` используйте `get-setf-expansion`, который введен на замену `get-setf-method-multiple-value`. Кроме того, продолжает использоваться `declare`, но больше не является оператором.

3. Были переименованы следующие четыре оператора:

```
define-setf-[method → expander]
get-setf-[method-multiple-value → expansion]
special-[form → operator]-p
simple-condition-format-[string → control]
```

   а также два типа (новых с 1990 года):

```
base-[character → char]
extended-[character → char]
```

4. Использование модулей, убранное в 1990 году, было восстановлено, однако объявлено нежелательным.
5. В качестве первого аргумента `setf` может быть использовано `values`-выражение.
6. Стандарт ANSI более конкретно определяет, какие функции могут принимать точечные списки. Например, сейчас определено, что то-

чечные списки могут быть аргументами nconc. (Странно, что аргументами append должны быть правильные списки, то есть nconc и append теперь принимают неодинаковые аргументы.)

7. Теперь невозможно преобразовывать integer в character с помощью coerce. Добавлена возможность преобразования (setf *f*) в функцию с помощью coerce.
8. Ослаблено ограничение, согласно которому аргумент compile должен быть определен в нулевом лексическом окружении; теперь окружение может содержать определения и декларации локальных макросов или символов-макросов. Также первым аргументом теперь может быть уже скомпилированная функция.
9. Функции gentemp и set признаны нежелательными.
10. Символ type в декларациях типов может быть опущен. Таким образом, ошибкой является определение типа с таким же именем, как и декларация, и наоборот.
11. Новая декларация ignorable может использоваться, чтобы избавиться от предупреждений, независимо от того, используется переменная или нет.
12. Константа array-total-size-limit теперь задана как тип fixnum. Поэтому аргумент row-major-aref можно всегда объявлять как fixnum.
13. Вместо :print-function структура, определяемая через defstruct, может иметь :print-object, который принимает только первые два аргумента.
14. Введен новый тип boolean, которому принадлежат только nil и t.

D

Справочник по языку

Это приложение описывает каждый оператор в ANSI Common Lisp. Мы будем следовать ряду соглашений:

Синтаксис

Описания функций представлены списком, начинающимся с имени функции, за которым следуют описания ее параметров. Описания специальных операторов и макросов являются регулярными выражениями, показывающими внешний вид корректного вызова.

В этих регулярных выражениях звездочка[1], следующая за объектом, означает, что объект повторяется ноль и более раз: `(a*)` может соответствовать `()`, `(a)`, `(a a)` и т. д. Объект в квадратных скобках соответствует его присутствию ноль или один раз: `(a [b] c)` может быть `(a c)` или `(a b c)`. Иногда для группировки используются фигурные скобки: `({a b}*)` может соответствовать `()`, `(a b)`, `(a b a b)` и т. д. Вертикальная черта означает выбор между несколькими альтернативами: `(a {1 | 2} b)` может быть `(a 1 b)` или `(a 2 b)`.

Имена параметров

Имена параметров соответствуют ограничениям на аргументы. Если параметр имеет имя, идентичное имени некоторого типа, то соответствующий аргумент может принадлежать только к этому типу. Ряд других имен (с описаниями) перечислен в таблице ниже.

Аргументы макросов и специальных операторов не вычисляются до тех пор, пока это не будет явно указано в описании. Если такой аргумент не вычисляется, ограничения на тип задаются его именем применительно

---

[1] Надстрочный знак * в регулярных выражениях не следует путать с *.

к самому аргументу, а если вычисляется, то применительно к его значению. Аргумент макроса вычисляется в том окружении, где встречается макровызов, если в описании явно не сказано другое.

alist Должен быть ассоциативным списком, т. е. правильным списком, состоящим из элементов вида *(key . value)*.

body Обозначает аргументы, которые могут следовать после списка параметров в выражении `defun`: либо *declaration** [*string*] *expression**, либо [*string*] *declaration** *expression**. Выражение вида (`defun` *fname parameters . body*) означает, что синтаксис выражения `defun` может быть (`defun` *fname parameters declaration** [*string*] *expression**) или (`defun` *fname parameters* [*string*] *declaration** *expression**). Если после строки *string* есть хотя бы одно выражение *expression*, то строка считается документацией.

c Комплексное число.

declaration Список, `car` которого — `declare`.

environment Показывает, что объект представляет собой лексическое окружение. (Создавать такие объекты вручную вы не можете, Лисп использует эти параметры для своих целей.) Символ `nil` всегда представляет глобальное окружение.

f Число с плавающей запятой.

fname Имя функции: либо символ, либо список (`setf` *s*), где *s* — символ.

format Строка, которая может быть вторым аргументом `format`, либо функция, которая принимает поток и ряд необязательных аргументов и (предположительно) записывает что-либо в этот поток.

i Целое число.

list Список любого типа. Может ли он быть циклическим, зависит от контекста. Функции, принимающие списки в качестве аргументов, могут работать с cdr-циклическими списками только в том случае, если их задача никогда не потребует поиска конца списка. Таким образом, `nth` может работать с cdr-циклическими списками, а `find` — нет. Параметр, названный *list*, всегда может быть car-циклическим.

n Неотрицательное целое число.

object Объект любого типа.

package Пакет, или строка, представляющая имя пакета, или символ, представляющий имя пакета.

path Может быть путем к файлу, потоком, связанным с файлом (в таком случае из него можно получить имя соответствующего файла), или строкой.

place Выражение, которое может быть первым аргументом `setf`.

plist Список свойств, т. е. правильный список с четным количеством элементов.

pprint-dispatch Таблица диспетчеризации красивой печати (или может быть `nil`).

predicate Функция.

prolist	Правильный список.
proseq	Правильная последовательность, т. е. вектор или правильный список.
r	Действительное число (`real`).
tree	Любой объект, являющийся деревом; не может быть car-или cdr-циклическим списком.
type	Спецификатор типа.

Умолчания

Некоторые необязательные параметры всегда имеют определенные значения по умолчанию. Необязательный параметр *stream* всегда по умолчанию имеет значение `*standard-input*` или `*standard-output*` в зависимости от направления потока. Необязательный параметр *package* по умолчанию установлен в `*package*`; *readtable* всегда установлен в `*readtable*`, а *pprint-dispatch* – в `*print-pprint-dispatch*`.

Сравнение

Многие функции, сравнивающие элементы последовательностей, принимают следующие аргументы по ключу: `key`, `test`, `test-not`, `from-end`, `start` или `end`. Они всегда используются одинаково (стр. 79). Параметры `key`, `test` и `test-not` должны быть функциями, `start` и `end` – неотрицательными целыми числами. В описании таких функций слова «сравнивает», «член» и «элемент» следует понимать с учетом влияния подобных аргументов.

В любой функции, сравнивающей элементы последовательности, предикатом сравнения по умолчанию является `eql`.

Структура

Если оператор возвращает структуру (например, список), необходимо помнить, что возвращаемое значение может разделять структуру с объектами, переданными в качестве аргументов, до тех пор, пока в описании оператора не будет явно указано, что это значение является вновь созданной структурой. Однако при вызове функции могут быть модифицированы лишь параметры, заключенные в угловые скобки (⟨*list*⟩). Если две функции перечисляются одна за другой, это значит, что вторая является деструктивным аналогом первой.

Вычисление и компиляция

(`compile` *fname* `&optional` *function*) функция

Если *function* не передана и *fname* является именем нескомпилированной функции или макроса, `compile` заменяет эту функцию или

макрос их скомпилированной версией, возвращая *fname*. (Лексическое окружение функции или макроса не должно отличаться от глобального окружения, за исключением определений локальных макросов, символовмакросов и деклараций.) Если *function* (которая может быть функцией или лямбда-выражением) передана, то `compile` преобразует ее к функции, компилирует и присваивает ей имя *fname*, возвращая его. *fname* также может быть `nil`, и в этом случае возвращается скомпилированная функция. Также возвращаются два дополнительных значения: второе является истинным, если при компиляции были получены ошибки или предупреждения; третье является истинным, если при компиляции были получены ошибки и предупреждения, отличные от предупреждений о стиле (style warnings).

(`compiler-macro-function` *fname* `&optional` *environment*) функция

Возвращает макрос компилятора, именем которого в окружении *environment* является *fname* (или `nil`, если ничего не найдено). Макрос компилятора – это функция двух аргументов: всего вызова и окружения, в котором этот вызов произошел. Может использоваться в `setf`.

(`constantp` *expression* `&optional` *environment*) функция

Возвращает истину, если *expression* является именем константы, или списком, `car` которого – `quote`, или объектом, который не является ни символом, ни cons-ячейкой. Также может возвращать истину для других выражений, которые считаются константами в используемой реализации.

(`declaim` *declaration-spec*) макрос

Аналогичен `proclaim`, но его вызов в toplevel обрабатывается компилятором, а *declaration-spec* не вычисляется.

(`declare` *declaration-spec**)

По поведению похож на специальный оператор, но не является таковым. Выражение, `car` которого – `declare`, может находиться в начале тела кода. Определяет декларации в соответствии с *declaration-spec* (не вычисляется) для всего кода в том окружении, где находится декларация. Допустимы следующие декларации: `dynamic-extent`, `ftype`, `ignorable`, `ignore`, `inline`, `notinline`, `optimize`, `special`, `type`.

(`define-compiler-macro` *fname parameters* . *body*) макрос

Похож на `defmacro`, но определяет макрос компилятора. Макросы компилятора похожи на обычные макросы, но раскрываются только компилятором и перед всеми обычными макросами. Обычно *fname* – это имя уже существующей функции или макроса. Макрос компиляции предназначен для оптимизации лишь некоторых вызовов, остальные же он оставляет без изменений. (Если бы нормальный макрос возвращал исходное выражение, это вызвало бы ошибку.) Строка документации становится документацией `compiler-macro` для имени *fname*.

(define-symbol-macro *symbol expression*) макрос

Заставляет расценивать обращение к *symbol* как макровызов, если еще существует специальная переменная с таким же именем *symbol*. Раскрытием этого макроса будет *expression*. Если *symbol* встречается в вызове setq, то setq будет вести себя как setf; аналогично multiple-value-setq будет вести себя как setf для values.

(defmacro *symbol parameters* . *body*) макрос

Глобально определяет макрос с именем *symbol*. В большинстве случаев можно считать, что выражение вида (*symbol* $a_1 \ldots a_n$) перед вычислением заменяется на значение, возвращаемое ((lambda *parameters* . *body*) $a_1 \ldots a_n$). В общем случае *parameters* связаны с аргументами макровызова, как и при использовании destructing-bind; они также могут содержать параметры &whole, который будет связан со всем макровызовом, и &environment, который будет связан с окружением, в котором встречен макровызов. Строка документации становится документацией function для имени *symbol*.

(eval *expression*) функция

Вычисляет выражение в глобальном окружении и возвращает его значение(я).

(eval-when (*case**) *expression**) специальный оператор

Если применимо хотя бы одно условие из *case*, выражения *expressions* вычисляются по порядку и возвращается значение последнего; в противном случае – nil. Условие (*case*) :compile-toplevel применимо, когда выражение eval-when находится на верхнем уровне файла при его компиляции. *Условие* :load-toplevel применимо, когда выражение eval-when попадает в toplevel при загрузке файла. *Условие* :execute применяется, когда выражение eval-when вычисляется любым доступным способом (поэтому только использование условия :execute делает вызов eval-then эквивалентным progn). Символы compile, load и eval являются нежелательными синонимами :compile-toplevel, :load-toplevel и :execute.

(lambda *parameters* . *body*) макрос

Эквивалентен (function (lambda *parameters* . *body*)).

(load-time-value *expression* &optional *constant*) специальный оператор

Эквивалентен (quote *val*), где *val* – значение, возвращаемое выражением *expression* при загрузке скомпилированного файла, содержащего выражение load-time-value. Если *constant* (не вычисляется) равна t, это указывает, что значение не будет изменено.

(locally *declaration** *expression**) специальный оператор

Выражение вида (locally $e_1 \ldots e_n$) эквивалентно (let () $e_1 \ldots e_n$), за исключением случая, когда эта форма находится в верхнем уровне;

тогда и выражения *expressions* считаются относящимися к верхнему уровню.

(macroexpand *expression* &optional *environment*) функция

Возвращает раскрытие макровызова *expression*, получаемое последовательным применением macroexpand-1 до тех пор, пока результат перестанет быть макровызовом. Второе возвращаемое значение истинно, если результат отличен от исходного *expression*.

(macroexpand-1 *expression* &optional *environment*) функция

Если *expression* является макровызовом, выполняет один этап его раскрытия, в противном случае возвращает *expression*. Принцип работы: вызывает *macroexpand-hook*, исходное значение которого – funcall, с тремя аргументами: соответствующая макрофункция, *expression* и *environment*. Таким образом, обычно раскрытие производится передачей аргументов macroexpand-1 в вызов макрофункции. Второе возвращаемое значение истинно, если результат отличен от исходного *expression*.

(macro-function *symbol* &optional *environment*) функция

Возвращает макрофункцию с именем *symbol* в окружении *environment* или nil в случае отсутствия такой функции. Макрофункция – это функция двух аргументов: самого вызова и окружения, в котором он происходит. Может использоваться в setf.

(proclaim *declaration-spec*) функция

Выполняет глобальные декларации согласно *declaration-spec*. Допустимы следующие декларации: declaration, ftype, inline, notinline, optimize, special, type.

(special-operator-p symbol) функция

Возвращает истину, если *symbol* является именем специального оператора.

(symbol-macrolet ((*symbol expression*)*) *declaration** *expression**) специальный оператор

Вычисляет свое тело, локально связав каждый символ *symbol* с соответствующим символом-макросом, таким же образом, как и с помощью define-symbol-macro.

(the *type expression*) специальный оператор

Возвращает значение выражения *expression* и декларирует его принадлежность к типу *type*. Может работать с выражениями, возвращающими несколько значений. (Спецификаторы типов с values описаны на стр. 418.) Количество деклараций и количество значений могут не совпадать: лишние декларации должны быть t или nil; для лишних значений типы не декларируются.

(quote *object*) специальный оператор

Возвращает свой аргумент без его вычисления.

Типы и классы

(coerce *object type*) функция

Возвращает эквивалентный объект типа *type*. Если объект *object* уже принадлежит типу *type*, возвращается он сам. В противном случае, если объект является последовательностью и последовательность типа *type* может содержать те же элементы, что и *object*, то возвращается последовательность типа *type* с теми же элементами, как *object*. Если объект – это строка из одного знака или символ, именем которого является строка из одного знака, и задан тип character, то возвращается данный знак. Если объект является действительным числом (real) и тип *type* соответствует числу с плавающей запятой, результатом будет аппроксимация *object* в виде числа с плавающей запятой. Если объект – это имя функции (символ, или (setf *f*), или лямбда-выражение) и задан тип function, то возвращается функция, на которую указывает объект; в случае лямбда-выражения функция будет определена в глобальном окружении, а не в окружении вызова coerce.

(deftype *name parameters* . *body*) макрос

Похож на defmacro, с той лишь разницей, что «вызов» *name* используется в качестве указателя типа (например, в рамках declare), а не выражения. Строка документации становится документацией типа type для имени *name*.

(subtypep *type1 type2* &optional *environment*) функция

Возвращает два значения: первое истинно тогда, когда может быть доказано, что *type1* является подтипом *type2*, второе – когда взаимоотношения между двумя типами известны точно.

(type-error-datum *condition*) функция

Возвращает объект, вызвавший ошибку типа *condition*.

(type-error-expected-type *condition*) функция

Возвращает тип, к которому мог принадлежать объект, вызвавший ошибку типа *condition*.

(type-of *object*) функция

Возвращает спецификатор типа, к которому принадлежит объект *object*.

(typep *object type* &optional *environment*) функция

Возвращает истину, если объект *object* принадлежит типу *type*.

Управление вычислением и потоками данных

(and *expression**) макрос

Вычисляет выражения по порядку. Если значение одного из них будет nil, дальнейшее вычисление прерывается и возвращается nil. Если все выражения истинны, возвращается значение последнего. Если выражения не заданы, возвращает t.

(apply *function* &rest *args*) функция

Применяет функцию *function* к аргументам *args*, которых должно быть не менее одного. Последний аргумент *arg* должен быть списком. Аргументами вызова *function* являются все элементы *args*, кроме последнего, плюс все элементы последнего элемента *args*, то есть список аргументов составляется так же, как с помощью list*. Функция *function* может быть также символом, и в этом случае используется связанная с ним глобальная функция.

(block *symbol expression**) специальный оператор

Вычисляет тело, заключенное в блок с именем *symbol* (не вычисляется). Используется вместе с return-from.

(case *object* (*key expression**)* [({t | otherwise} *expression**)]) макрос

Вычисляет объект, затем сверяет его значение с последующими выражениями по такой схеме: если значение равно (eql) ключу *key* (не вычисляется) или принадлежит (member) ему, то вычисляются соответствующие ключу выражения и возвращается значение последнего из них. Ключи t и otherwise соответствуют любому значению. Если совпадение не найдено, возвращается nil. Символы t и otherwise не могут использоваться в качестве ключей, но тот же эффект можно получить, используя (t) и (otherwise).

(catch *tag expression**) специальный оператор

Вычисляет свое тело, добавляя перехват метки, значение которой вычисляется в *tag*. Используется совместно с throw.

(ccase *object* (*key expression**)*) макрос

Действует аналогично case, за исключением случая, когда совпадение не найдено. Если совпадение не найдено, сигнализирует о корректируемой ошибке типа.

(compiled-function-p *object*) функция

Возвращает истину, если *object* является скомпилированной функцией.

(complement *predicate*) функция

Возвращает функцию одного аргумента, которая возвращает истину, когда заданный предикат *predicate* (также принимающий один

аргумент) возвращает ложь, и наоборот, возвращает ложь, если *predicate* возвращает истину.

(cond (*test expression**)*) макрос

Вычисляет выражения *test* до тех пор, пока одно из них не окажется истинным. Если с ним не связаны какие-либо выражения, возвращается значение его самого, в противном случае связанные с ним выражения вычисляются по очереди и возвращается значение последнего. Если ни одно тестовое выражение не оказалось истинным, возвращается nil.

(constantly *object*) функция

Возвращает функцию, принимающую произвольное количество аргументов; также возвращает *object*.

(ctypecase *object* (*type expression**)*) макрос

Вычисляет объект, затем по очереди проверяет, относится ли он к типам *type*, заданным в следующих выражениях (не вычисляются). Если найдено совпадение, вычисляются соответствующие выражения *expressions* и возвращается значение(я) последнего из них. В случае отсутствия связанных выражений возвращается nil. В случае отсутствия совпадения сигнализируется о корректируемой ошибке типа.

(defconstant *symbol expression* [*string*]) макрос

Определяет символ как глобальную константу со значением выражения *expression*. Никакие глобальные или локальные переменные не могут иметь такое же имя. Выражение *expression* может вычисляться на этапе компиляции. Строка *string*, если задана, становится строкой документации variable для имени *symbol*. Возвращает *symbol*.

(define-modify-macro *name parameters symbol* [*string*]) макрос

Выражение вида (define-modify-macro *m* ($p_1 \ldots p_n$) *f*) определяет новый макрос *m*, такой что вызов вида (*m place* $a_1 \ldots a_n$) установит значение по месту *place* в (*f val* $a_1 \ldots a_n$), где *val* содержит текущее значение *place*. Среди параметров могут быть также необязательные и остаточные. Строка *string*, если задана, становится документацией нового макроса.

(define-setf-expander *reader parameters* . *body*) макрос

Определяет порядок раскрытия выражений вида (setf (*reader* $a_1 \ldots a_n$) *val*). При вызове get-setf-expansion для такого выражения будут возвращены пять значений. Строка *string*, если задана, становится документацией setf для символа *reader*.

(defparameter *symbol expression* [*string*]) макрос

Присваивает глобальной переменной *symbol* значение выражения *expression*. Строка *string*, если задана, становится документацией variable для *symbol*. Возвращает *symbol*.

(defsetf *reader writer* [*string*]) макрос

Сокращенный вид: преобразует вызовы вида (setf (*reader* a_1...a_n) *val*) в (*writer* a_1...a_n *val*); *reader* и *writer* должны быть символами и не вычисляются. Строка *string*, если задана, становится документацией setf для символа *reader*.

(defsetf *reader parameters* (*var**) . *body*) макрос

Расширенный вид: преобразует вызовы вида (setf (*reader* a_1...a_n) *val*) в выражения, получаемые вычислением тела defsetf, как если бы это был defmacro. *reader* должен быть символом (не вычисляется), соответствующим имени функции или макроса, вычисляющего все свои аргументы; *parameters* – это список параметров для *reader*, а *vars* будут представлять значение(я) *val*. Строка *string*, если задана, становится документацией setf для символа *reader*.

Чтобы соответствовать принципу, по которому setf возвращает новое значение своего первого аргумента, раскрытие defsetf также должно возвращать это значение.

(defun *fname parameters* . *body*) макрос

Глобально определяет *fname* как функцию, определенную в лексическом окружении, в котором находится тело defun. Тело функции заключено в блок с именем *fname*, если *fname* – символ, или *f*, если *fname* – список вида (setf *f*). Строка *string*, если задана, становится документацией function для *fname*.

(defvar *symbol* [*expression* [*string*]]) макрос

Присваивает глобальной переменной *symbol* значение *expression*, если выражение *expression* задано и эта переменная еще не имела значения. Строка *string*, если задана, становится документацией variable для *symbol*. Возвращает *symbol*.

(destructing-bind *variables tree declaration* expression**) макрос

Вычисляет свое тело для набора переменных *variables* (дерево с внутренними узлами, которые являются списками аргументов), связанных со значениями, соответствующими элементам дерева *tree*. Структура двух деревьев должна совпадать.

(ecase *object* (*key expression**)*) макрос

Подобен ccase, но вызывает некорректируемую ошибку типа при отсутствии совпадения.

(eq *object1 object2*) функция

Возвращает истину, если *object1* и *object2* идентичны.

(eql *object1 object2*) функция

Возвращает истину, если *object1* и *object2* равны с точки зрения eq или являются одним и тем же знаком или числами, выглядящими одинаково при печати.

(equal *object1 object2*) функция

Возвращает истину, если *object1* и *object2* равны с точки зрения eql; либо являются cons-ячейками, чьи car и cdr эквивалентны с точки зрения equal; либо являются строками или бит-векторами одной длины (с учетом указателей заполнения), чьи элементы эквивалентны с точки зрения eql; либо являются путями, компоненты которых эквивалентны. Для циклических аргументов вызов может не завершиться.

(equalp *object1 object2*) функция

Возвращает истину, если *object1* и *object2* равны с точки зрения equal, char-equal или =; либо являются cons-ячейками, car и cdr которых эквивалентны с точки зрения equalp; либо являются массивами одинаковой длины, элементы которых эквивалентны с точки зрения equalp; либо являются структурами одного типа, элементы которых равны с точки зрения equalp; либо являются хеш-таблицами с одинаковыми тестовыми функциями и количеством элементов, ключи которых связаны со значениями, равными для двух таблиц с точки зрения equalp. Логично предполагать, что вызов с циклическими аргументами может не завершиться.

(etypecase *object* (*key exression**)*) макрос

Подобен ctypecase, но вызывает некорректируемую ошибку типа, если не находит совпадения с одним из ключей *key*.

(every *predicate proseq* &rest *proseqs*) функция

Возвращает истину, если предикат *predicate*, который должен быть функцией стольких аргументов, сколько последовательностей передано, истинен для всех первых элементов последовательностей *proseqs*, затем для всех вторых элементов, и так до *n*-го элемента, где *n* – длина кратчайшей последовательности *proseq*. Проход по последовательностям завершается, когда предикат возвратит nil. В этом случае в результате всего вызова возвращается nil.

(fboundp *fname*) функция

Возвращает истину, если *fname* является именем глобальной функции или макроса.

(fdefinition *fname*) функция

Возвращает глобальную функцию с именем *fname*. Может использоваться в setf.

(flet ((*fname parameters* . *body*)*) специальный оператор
declaration expression**)

Вычисляет свое тело, связав локально каждое имя *fname* с соответствующей функцией. Действует подобно labels, но локальные функции видны лишь внутри тела; они не могут вызывать сами себя и друг друга (и поэтому не могут быть рекурсивными).

(fmakunbound *fname*) функция

Удаляет определение глобальной переменной или макроса для *fname*. Если определение не найдено, вызывает ошибку. Возвращает *fname*.

(funcall *function* &rest *args*) функция

Применяет функцию *function* к аргументам *args*. Функция может быть задана как символ, и в этом случае используется определение глобальной функции, связанной с этим символом.

(function *name*) специальный оператор

Возвращает функцию с именем *name*, которое может быть символом, списком вида (setf *f*) или лямбда-выражением. Если *f* – встроенный оператор, наличие или отсутствие функции (setf *f*) зависит от используемой реализации.

(function-lambda-expression *function*) функция

Предназначена для получения лямбда-выражения, соответствующего функции *function*, но может также возвращать nil. Возвращает два дополнительных значения: второе, будучи nil, говорит о том, что функция была определена в нулевом лексическом окружении; третье значение может использоваться вашей реализацией, чтобы вернуть имя *function*.

(functionp *object*) функция

Возвращает истину, если *object* является функцией.

(labels ((*fname parameters* . *body*)*) специальный оператор
declaration expression**)

Вычисляет свое тело, предварительно связав локально имена *fname* с соответствующими определениями функций. Действует подобно flet, но имена локальных функций видны внутри самих определений. Такие функции могут вызывать друг друга и самих себя (то есть могут быть рекурсивными).

(get-setf-expansion *place* &optional *environment*) функция

Возвращает пять значений ($v_1 ... v_5$), определяющих вид раскрытия выражений (setf *place val*) в окружении *environment*. Эти пять значений представляют собой: список уникальных имен переменных (gensym); такого же размера список значений, которые должны быть им присвоены; список временных переменных, в которые будут записаны новые значения для *place*; выражение, выполняющее установление нового значения, которое может ссылаться на переменные из v_1 и v_3; выражение, которое будет извлекать оригинальное значение *place* и может ссылаться на переменные из v_1.

(go *tag*) специальный оператор

Находясь внутри выражения tagbody, передает управление по адресу ближайшего тега из лексической области видимости, равному заданному с точки зрения eql.

(identity *object*) функция

Возвращает объект *object*.

(if *test then* [*else*]) специальный оператор

Вычисляет значение *test*. Если оно истинно, вычисляет и возвращает значение выражения *then*; в противном случае вычисляет и возвращает значение *else* или возвращает nil в случае отсутствия *else*.

(let ({*symbol* | (*symbol* [*value*])}*) специальный оператор
*declaration** *expression**)

Вычисляет свое тело, предварительно связав каждый символ с соответствующим значением или с nil в случае отсутствия значения *value*.

(let* ({*symbol* | (*symbol* [*value*])}*) специальный оператор
*declaration** *expression**)

Действует подобно let, с той лишь разницей, что выражения *value* могут ссылаться на предыдущие символы *symbol*.

(macrolet ((*symbol parameters* . *body*)*) специальный оператор
*declaration** *expression**)

Вычисляет свое тело, предварительно локально связав каждый символ с соответствующим макросом. Функции раскрытия определяются в том лексическом окружении, где находится вызов macrolet. Подобно flet, локальные макросы не могут вызывать друг друга.

(multiple-value-bind (*symbol**) *expression1* макрос
*declaration** *expression**)

Вычисляет *expression1*, затем вычисляет его тело, предварительно связав каждый символ *symbol* (не вычисляется) с соответствующим возвращаемым значением *expression1*. Если значений меньше, чем символов, остальные символы получают значения nil; если символов меньше, чем значений, то лишние значения игнорируются.

(multiple-value-call *function expression**) специальный оператор

Вызывает функцию *function* (вычисляется) с аргументами, которыми являются все значения всех выражений *expression*.

(multiple-value-list *expression*) макрос

Возвращает список значений, возвращаемых выражением *expression*.

(multiple-value-prog1 *expression1 expression**) специальный оператор

Вычисляет свои аргументы друг за другом, возвращая значение(я) первого.

(multiple-value-setq (*symbol**) *expression*) макрос

Связывает символы (они не вычисляются) со значениями, возвращаемыми выражением *expression*. Если значений слишком мало, оставшиеся символы получают nil; если значений слишком много, лишние игнорируются.

(not *object*) функция

Возвращает истину, если объект *object* имеет значение nil.

(notany *predicate proseq* &rest *proseqs*) функция

Выражение вида (notany *predicate* $s_1 \ldots s_n$) эквивалентно (not (some *predicate* $s_1 \ldots s_n$)).

(notevery *predicate proseq* &rest *proseqs*) функция

Выражение вида (notevery *predicate* $s_1 \ldots s_n$) эквивалентно (not (every *predicate* $s_1 \ldots s_n$)).

(nth-value *n expression*) макрос

Возвращает *n*-е (вычисляется) значение выражения *expression*. Нумерация начинается с 0. В случае когда выражение вернуло меньше *n*+1 значений, возвращается nil.

(or *expression**) макрос

Вычисляет выражения друг за другом до тех пор, пока одно из значений не окажется истинным. В таком случае возвращается само значение, в противном случае – nil. Может возвращать множественные значения, но только из последнего выражения.

(prog ({*symbol* | (*symbol* [*value*])}*)
*declaration** {*tag* | *expression*}*) макрос

Вычисляет свое тело, предварительно связав каждый символ со значением соответствующего выражения *value* (или с nil, если ничего не задано). Тело заключено в tagbody и блок с именем nil, поэтому выражения *expressions* могут использовать go, return и return-from.

(prog* ({*symbol* | (*symbol* [*value*])}*)
*declaration** {*tag* | *expression*}*) макрос

Действует подобно prog, но выражения *value* могут использовать ранее связанные символы.

(prog1 *expression1 expression**) макрос

Вычисляет свои аргументы один за другим, возвращая значение первого.

(prog2 *expression1 expression2 expression**) макрос

Вычисляет свои аргументы один за другим, возвращая значение второго.

(progn *expression**) специальный оператор

Вычисляет свои аргументы один за другим, возвращая значение(я) последнего.

(progv *symbols values expression**) специальный оператор

Вычисляет свое тело, предварительно связав динамически каждый символ в списке *symbols* с соответствующим элементом в списке

values. Если переменных слишком много, попытка сослаться на оставшиеся вызовет ошибку; если переменных слишком мало, лишние значения будут проигнорированы.

(psetf {*place value*}*) макрос

Действует подобно setf, но если выражение *value* ссылается на одно из предшествующих мест *place*, то оно получит предыдущее значение. Это значит, что (psetf *x y y x*) поменяет значения *x* и *y*.

(psetq {*symbol value*}*) макрос

Действует подобно psetf, но оперирует символами, а не местами.

(return *expression*) макрос

Эквивалентен (return-from nil *expression*). Многие макросы (в том числе и do) неявно заключают свое тело в блок с именем nil.

(return-from *symbol expression*) специальный оператор

Возвращает значение(я) выражения *expression* из лексически ближайшего блока с именем *symbol* (не вычисляется). Попытка использования return-from не внутри блока с заданным именем приводит к ошибке.

(rotatef *place**) макрос

Смещает значения своих аргументов на одно влево, как в циклическом буфере. Все аргументы вычисляются по очереди, а затем, если вызов имел вид (rotatef $a_1 \ldots a_n$), место a_2 получает значение места a_1, a_3 – значение a_2 и т. д., a_1 получает значение a_n. Возвращает nil.

(setf {*place value*}*) макрос

Обобщение setq: помещает значение выражения *value* по заданному месту. Если выражение *value* ссылается на одно из предыдущих мест *places*, оно будет использовать новое значение этого места. Возвращает значение последнего выражения *value*.

Корректным выражением для места *place* может быть: переменная; вызов любой «устанавливаемой» функции при условии, что соответствующий аргумент является корректным выражением для *place*; вызов apply с первым аргументом из числа следующих: #'aref, #'bit или #'sbit; вызов функции доступа к полям структуры; выражение the или values, аргумент(ы) которого являются корректными местами *places*; вызов оператора, для которого задано setf-раскрытие; или макрос, раскрывающийся в что-либо из вышеперечисленного.

(setq {*symbol value*}*) специальный оператор

Присваивает каждой переменной *symbol* значение соответствующего выражения *value*. Если одно из выражений ссылается на ранее определенную переменную, оно будет использовать новое значение. Возвращает значение последнего выражения *value*.

(shiftf *place1 place* expression*) макрос

Смещает значения своих аргументов на одно влево. Аргументы вычисляются один за другим; затем, если вызов имел вид (shiftf $a_1 \ldots a_n$ *val*), значение a_2 помещается в место a_1 и т. д., а в место a_n помещается значение *val*. Возвращается значение a_1.

(some *predicate proseq* &rest *proseqs*) функция

Возвращает истину, если предикат *predicate*, который должен быть функцией стольких аргументов, сколько задано последовательностей *proseqs*, возвращает истину, будучи примененным к первым элементам всех последовательностей, или ко вторым, или к *n*-м, где *n* – длина кратчайшей последовательности *proseq*. Проход по последовательностям останавливается тогда, когда предикат вернет истину, возвращая при этом значение предиката на тот момент.

(tagbody {*tag* | *expression*}*) специальный оператор

Вычисляет выражения *expressions* одно за другим и возвращает nil. Может содержать вызовы go, изменяющие порядок вычисления выражений. Теги *tags*, которые должны быть символами или целыми числами, не вычисляются. Атомы, полученные в результате раскрытия макросов, тегами не считаются. Все макросы, имена которых начинаются с do, а также prog и prog*, неявно заключают свое тело в вызов tagbody.

(throw *tag expression*) специальный оператор

Возвращает значение(я) выражения *expression* из ближайшего в динамической области видимости выражения catch с такой же меткой (eq), как и *tag*.

(typecase *object* (*type expression**)*) макрос
[({t | otherwise} *expression**)])

Вычисляет объект *object*, затем по очереди сверяет его значение с перечисленными типами *type* (не вычисляются). Если найдено совпадение или достигнут вариант t либо otherwise, вычисляются соответствующие выражения и возвращается значение последнего из них. Возвращает nil, если совпадений не найдено или для совпавшего типа не определено никаких выражений.

(unless *test expression**) макрос

Выражение вида (unless *test* $e_1 \ldots e_n$) эквивалентно (when (not *test*) $e_1 \ldots e_n$).

(unwind-protect *expression1 expression**) специальный оператор

Вычисляет свои аргументы по порядку и возвращает значение(я) первого. Остальные выражения будут вычислены, даже если вычисление первого было прервано.

(values &rest *objects*) функция

Возвращает свои аргументы.

(values-list *prolist*) функция

Возвращает элементы списка *prolist*.

(when *test expression**) макрос

Вычисляет выражение *test*. Если оно истинно, по порядку вычисляет выражения *expressions* и возвращает значение(я) последнего; в противном случае возвращает nil. Возвращает nil, если не задано ни одного выражения *expression*.

Итерация

(do ({*var* | (*var* [*init* [*update*]])}*) макрос
(*test result**)
*declaration** {*tag* | *expression*}*)

Вычисляет свое тело сначала для исходных значений переменных *var* (если исходное значение переменной *init* не задано, используется nil), а затем для новых значений переменных, полученных из выражений *update* (или для предыдущего значения, если *update* не задано). Каждый раз вычисляется выражение *test*; если значение *test* ложно, вычисляется тело цикла, если *test* истинно, цикл завершается вычислением выражений *result* по порядку, и значение последнего из них возвращается. Тело цикла неявно заключено в tagbody, а выражение do целиком заключено в блок с именем nil.

Если выражение *init* ссылается на переменную с тем же именем, что и у *var*, то будет использоваться ее текущее значение в контексте, где находится выражение do. Если на переменную *var* ссылается выражение *update*, будет использоваться значение из предыдущей итерации. Таким образом, значения устанавливаются так же, как с помощью let, а изменяются так же, как при использовании psetq.

(do* ({*var* | (*var* [*init* [*update*]])}*) макрос
(*test result**)
*declaration** {*tag* | *expression**}*)

Действует подобно do, но значения переменных устанавливаются последовательно, как с помощью let*, а изменяются так же, как при использовании setq.

(dolist (*var list* [*result*]) макрос
*declaration** {*tag* | *expression*}*)

Вычисляет свое тело, связывая переменную *var* последовательно с каждым элементом списка *list*. Если передан пустой список, тело не будет вычислено ни разу. Тело неявно заключено в tagbody, вызов dolist целиком заключен в блок с именем nil. Возвращает значение(я) выражения *result*; если это выражение отсутствует, возвращает nil.

Выражение *result* может ссылаться на переменную *var*, для которой будет использовано значение `nil`.

(`dotimes` (*var integer* [*result*]) макрос
*declaration** {*tag* | *expression*}*)

Вычисляет свое тело, связывая переменную *var* последовательно с целыми числами от `0` до значения (`-` *integer* `1`) включительно. Если значение *integer* не является положительным, тело не будет вычислено ни разу. Тело неявно заключено в `tagbody`, вызов `dotimes` целиком заключен в блок с именем `nil`. Возвращает значение(я) выражения *result* либо `nil`, если это выражение отсутствует. Выражение *result* может ссылаться на переменную *var*, значение которой будет равно количеству вычислений тела цикла.

(`loop` *expression**) макрос

Сокращенная форма: если выражения *expression* не включают ключевых слов макроса `loop`, то они просто вычисляются друг за другом бесконечное число раз. Выражения неявно заключены в блок с именем `nil`.

(`loop` [*name-clause*] *var-clause* body-clause**) макрос

Развернутая форма: `loop`-выражение содержит ключевые слова, которые обрабатываются следующим образом:

1. Изначально все создаваемые переменные связаны с неопределенными, вероятно, случайными значениями корректного типа.
2. Вычисляется loop-пролог.
3. Переменные получают исходные значения.
4. Выполняются проверки на выход из цикла.
5. Если не удовлетворяется ни одно из условий выхода, вычисляется тело цикла, затем значения переменных обновляются и управление передается на предыдущий шаг.
6. Если одно из условий выхода выполняется, вычисляется эпилог и возвращаются заданные значения.

Отдельно взятое предложение может вносить свой вклад в несколько различных шагов. На каждом шаге предложения вычисляются в том порядке, в котором они встречаются в исходном коде.

Выражение `loop` всегда неявно заключено в блок. По умолчанию его имя — `nil`, но оно может быть задано с помощью предложения `named`.

Значением всего `loop`-выражения является последнее накопленное значение или `nil`, если такового не имеется. Накопленным значением можно считать значение, полученное во время итерации предложениями `collect`, `append`, `nconc`, `count`, `sum`, `maximize`, `minimize`, `always`, `never` или `thereis`.

Предложение *name-clause* задает имя блока с помощью `named`; *var-clause* может содержать предложения `with`, `initially`, `finally` или `for`;

body-clause может быть предложением любого типа, за исключением named, with или for. Каждое из этих предложений описывается ниже.

Соглашения: Элементы предложений не вычисляются до тех пор, пока в описании об этом не будет сказано явно. Слово *type* означает декларацию типа; это может быть отдельный спецификатор типа или дерево спецификаторов, которые связываются с соответствующими элементами списка по деструктуризации.

Синонимы: Макрос loop активно использует синонимы многих ключевых слов. В следующих парах ключей второй является полноценной заменой первому: upfrom, from; downfrom, from; upto, to; downto, to; the, each; of, in; when, if; hash-keys, hash-key; hash-value, hash-values; symbol, symbols; present-symbol, present-symbols; external-symbol, external-symbols; do, doing; collect, collecting; append, appending; nconc, nconcing; count, counting; sum, summing; maximize, maximizing; minimize, minimizing.

named *symbol*

Делает заданный символ *symbol* именем блока, в который заключается loop-выражение.

with *var1* [*type1*] = *expression1* {and *var* [*type*] = *expression*}*

Связывает каждую переменную *var* со значением соответствующего выражения *expression*, делая это параллельно, как с помощью let.

initially *expression1 expression**

Вычисляет выражения по порядку как часть loop-пролога.

finally *expression1 expression**

Вычисляет выражения по порядку как часть loop-эпилога.

for *var* [*type1*] *for-rest1* {and *var* [*type*] *for-rest*}*

Связывает каждую переменную *var* со значением, получаемым из соответствующего *for-rest* на каждой последующей итерации. Использование нескольких for-предложений, связанных через and, приведет к параллельному обновлению значений переменных, как в do. Возможные формы для *for-rest* выглядят следующим образом:

[upfrom *start*] [{upto | below} *end*] [by *step*]

Как минимум одно из этих трех подвыражений должно быть задействовано. Если используется несколько, то они могут следовать в произвольном порядке. Значение переменной исходно установлено в *start* или 0. На каждой последующей итерации оно увеличивается на *step* или 1. Выражение upto или below позволяет ограничить количество итераций: upto остановит итерации, когда значение переменной будет больше *end*, below – когда больше или равно. Если синонимы from и to используются совместно или же from используется без to, то from будет интерпретирован как upfrom.

`downfrom` ***start*** [{`downto` | `above`} ***end***] [`by` ***step***]

Как и выше, подвыражения могут следовать в произвольном порядке. Значение переменной исходно установлено в *start*. На каждой последующей итерации оно уменьшается на *step* или `1`. Выражение `downto` или `above` позволяет ограничить количество итераций: `downto` остановит итерации, когда значение переменной будет меньше *end*, `above` – когда меньше или равно.

{`in` | `on`} ***list*** [`by` ***function***]

Если первым ключевым словом является `in`, переменная получает по очереди все элементы списка *list*, если первым ключевым словом является `on`, переменная последовательно получает все хвосты. Если передана функция *function*, она применяется к списку вместо `cdr` на каждой итерации. Также добавляет тест на завершение: итерация прекращается по достижении конца списка.

`=` ***expression1*** [`then` ***expression2***]

Переменная исходно получает значение *expression1*. Если задано `then`, на каждой последующей итерации она будет получать значение *expression2*, иначе – *expression1*.

`across` ***vector***

Переменная связывается по очереди со всеми элементами вектора. Также добавляет тест на завершение: итерация завершается по достижении последнего элемента.

`being the hash-keys of` ***hash-table*** [`using (hash-value` ***v2***`)`]
`being the hash-values of` ***hash-table*** [`using (hash-key` ***v2***`)`]

В первом случае переменная будет связываться с одним из ключей хеш-таблицы (порядок связывания не определен), а *v2* (если задан) – с соответствующим ключу значением. Во втором случае переменная получит значение, а *v2* – соответствующий ему ключ. Добавляет тест на завершение: итерации прекратятся по достижении последнего ключа в хеш-таблице.

`being each` {`symbol` | `present-symbol` | `external-symbol`} [`of` ***package***]

В зависимости от ключа переменная будет получать все доступные символы (`symbol`), имеющиеся символы (`present-symbol`) или внешние символы (`external-symbol`) для пакета. Аргумент *package* используется так же, как в `find-package`: если он не задан, используется текущий пакет. Добавляет тест на завершение: итерация прекращается по достижении последнего символа.

`do` ***expression1 expression****

Вычисляет выражения одно за другим.

`return` ***expression***

Заставляет `loop`-выражение немедленно прекратить вычисления и возвратить значение *expression* без вычисления `loop`-эпилога.

{collect | append | nconc} *expression* [into *var*]

Собирает список (исходно nil) в процессе итерации. Если первое ключевое слово – collect, то значения выражения *expression* собираются в список; если append, то возвращаемые из *expression* значения объединяются в список с помощью append; если nconc, то делается то же самое деструктивно. Если предоставляется переменная *var*, то ей будет присвоен полученный список, но в этом случае он не будет возвращаться как результат loop-вызова по умолчанию.

Внутри loop предложения collect, append и nconc могут аккумулировать значения в одну и ту же переменную. Если для нескольких предложений не заданы отдельные переменные *var*, это будет расценено как намерение собирать все значения в один список.

{count | sum | maximize | minimize} *expression* [into *var*] [*type*]

Аккумулирует число в процессе итерации. Если используется ключевое слово count, возвращаемое значение будет отражать количество истинных результатов вычисления выражения *expression*; если sum, то будет возвращена сумма всех полученных значений; если maximize/minimize, то максимальное/минимальное значение из всех полученных. Для count и sum исходным значением будет 0, для maximize и minimize исходное значение не определено. Если задана *var*, она будет связана с полученным значением, и оно не будет возвращаться как значение loop-выражения по умолчанию. Также может быть задан тип *type* аккумулируемого значения.

Внутри loop предложения sum и count могут аккумулировать значения в одну и ту же переменную. Если для нескольких предложений не заданы отдельные переменные *var*, это будет расценено как намерение аккумулировать все значения в одно. Аналогично для maximize и minimize.

when *test then-clause1* {and *then-clause*}*
[else *else-clause1* {and *else-cause*}*] [end]

Вычисляет выражение *test*. Если оно истинно, по порядку вычисляются выражения *then-clause*, в противном случае – *else-clause*. Выражения *then-clause* и *else-clause* могут использовать do, return, when, unless, collect, append, nconc, count, sum, maximize и minimize.

Выражение в *then-clause1* или *else-clause1* может использовать it, в этом случае оно будет ссылаться на значение выражения *test*.

unless *test then-clause1* {and *then-clause*}*
[else *else-clause1* {and *else-clause*}*] [end]

Предложение вида unless *test* $e_1 \ldots e_n$ эквивалентно when (not *test*) $e_1 \ldots e_n$.

repeat *integer*

Добавляет тест на завершение: итерации прекращаются после выполнения *integer* итераций.

`while` *expression*

Добавляет тест на завершение: итерации прекращаются, когда выражение *expression* будет ложным.

`until` *expression*

Эквивалент `while (not` *expression*`)`.

`always` *expression*

Действует подобно `while`, но также определяет возвращаемое значение `loop`: `nil`, если *expression* ложно, в противном случае – `t`.

`never` *expression*

Эквивалент `always (not` *expression*`)`.

`thereis` *expression*

Действует подобно `until`, но также определяет возвращаемое значение `loop`: `t`, если *expression* ложно, `nil` – в противном случае.

`(loop-finish)` макрос

Может использоваться только внутри `loop`-выражения, где он завершает текущую итерацию и передает управление в `loop`-эпилог, после чего `loop`-выражение завершается нормально.

Объекты

`(add-method` *generic-function method*`)` обобщенная функция

Создает метод *method* для обобщенной функции *generic-function*, возвращая *generic-function*. При совпадении спецификаторов перезаписывает существующий метод. Добавляемый метод *method* не может принадлежать другой обобщенной функции.

`(allocate-instance` *class* `&rest` *initargs* `&key)` обобщенная функция

Возвращает экземпляр класса *class* с неинициализированными слотами. Допускает использование других ключей.

`(call-method` *method* `&optional` *next-methods*`)` макрос

При вызове внутри метода вызывает метод *method*, возвращая значения, которые вернет этот вызов. *next-method*, если передан, должен быть списком методов, которые будут вызваны следующими за *method*. Вместо метода *method* может также предоставляться список вида `(make-method` *expression*`)`, где *expression* – тело метода. В таком случае *expression* вычисляется в глобальном окружении, за исключением варианта, когда имеется локальное определение макроса `call-method`.

`(call-next-method &rest` *args*`)` функция

При вызове внутри метода вызывает следующий метод с аргументами *args*, возвращая значения его вызова. Если не предоставлено никаких аргументов *args*, используются аргументы вызова текущего

метода (игнорируя любые присваивания значений этим аргументам). В стандартной комбинации методов `call-next-method` может быть вызван в первичном или `around`-методе. Если следующий метод отсутствует, вызывает `no-next-method`, который по умолчанию сигнализирует об ошибке.

(`change-class` *instance class* `&rest` *initargs* `&key`) обобщенная функция

Изменяет класс экземпляра *instance* на класс *class*, возвращая *instance*. Если существующие слоты имеют те же имена, что и локальные слоты класса *class*, то они останутся нетронутыми, в противном случае эти слоты будут отброшены. Новые локальные слоты, требуемые классом *class*, инициализируются вызовом `update-instance-for-redefined-class`. Допускает использование других ключей.

(`class-name` *class*) обобщенная функция

Возвращает имя класса *class*. Может быть первым аргументом `setf`.

(`class-of` *object*) функция

Возвращает класс, экземпляром которого является *object*.

(`compute-applicable-methods` *generic-function args*) обобщенная функция

Возвращает отсортированный по убыванию специфичности список методов для обобщенной функции *generic-function*, вызванной с аргументами из списка *args*.

(`defclass` *name* (*superclass**) (*slot-spec**) *class-spec**) макрос

Определяет и возвращает новый класс с именем *name*. Если *name* уже является именем класса, то существующие экземпляры будут приведены к новому классу. Суперклассы, если они есть, задаются именами *superclass*, но их наличие не обязательно. Слотами нового класса является комбинация наследуемых слотов и локальных слотов, определенных в *slot-spec*. За объяснением механизма разрешения конфликтов обратитесь к стр. 428.

Каждое определение слота *slot-spec* должно быть символом *symbol* либо списком (*symbol* k_1 v_1*...k_n v_n*), где любой ключ *k* не может использоваться дважды. Символ *symbol* является именем слота. Ключи *k* могут быть следующими:

`:reader` *fname**

Для каждого *fname* определяет неквалифицированный метод с именем *fname*, возвращающий значение соответствующего слота.

`:writer` *fname**

Для каждого *fname* определяет неквалифицированный метод с именем *fname*, ссылающийся на соответствующий слот и пригодный для использования в `setf` (но этот метод не может быть вызван напрямую).

`:accessor` *fname**

Для каждого *fname* определяет неквалифицированный метод с именем *fname*, ссылающийся на соответствующий слот. Может использоваться в `setf`.

`:allocation` *where*

Если *where* – это `:instance` (по умолчанию), каждый экземпляр будет иметь собственный слот; если `:class`, то слот будет использоваться всеми экземплярами класса.

`:initform` *expression*

Если при создании экземпляра не передаются исходные значения слота и для него нет значения по умолчанию, значение слота будет установлено в значение *expression*.

`:initarg` *symbol**

Каждый символ *symbol* может использоваться как аргумент по ключу для задания значения слота при создании экземпляра с помощью `make-instance`. Символы не обязательно должны быть ключевыми словами. Один и тот же символ может быть использован как `initarg` в нескольких слотах класса.

`:type` *type*

Декларирует тип *type* значения, содержащегося в слоте.

`:documentation` *string*

Предоставляет строку документации к слоту.

Определения *class-spec* могут быть комбинацией выражений: (`:documentation` *string*), (`:metaclass` *symbol*) или (`:default-initargs` $k_1\ e_1 \ldots k_n\ e_n$). Последнее используется при создании экземпляров; см. `make-instance`. В любом вызове `make-instance` при отсутствии явного значения ключа *k* будет использоваться значение соответствующего выражения *e*, которое вычисляется. Параметр `:documentation` становится документацией к классу, а `:metaclass` может использоваться для задания метакласса, отличного от `standard-class`.

(`defgeneric` *fname parameters entry**) макрос

Определяет или дополняет текущее определение обобщенной функции *fname*. Возвращает саму функцию. Вызывает ошибку, если *fname* является обычной функцией или макросом.

parameters – это специализированный лямбда-список; см. `defmethod`. Все методы для обобщенной функции должны иметь списки параметров, конгруэнтные друг другу и списку *parameters*.

entry может включать в себя одну или несколько следующих опций:

(`:argument-precedence-order` *parameter**)

Переопределяет порядок предшествования, определяемый по умолчанию вторым аргументом `defgeneric`. Должны быть заданы все параметры обобщенной функции.

(declare (optimize *property**))

Декларация учитывается в процессе компиляции самой обобщенной функции, то есть кода, отвечающего за диспетчеризацию. Не влияет на компиляцию самих методов. См. declare.

(:documentation *string*)

Задает документацию к имени *fname* с ключом function.

(:method-combination *symbol arguments**)

Определяет тип комбинации методов, используемый для имени *symbol*. Встроенные типы комбинации не требуют каких-либо аргументов, собственные типы комбинации могут быть определены с помощью развернутой формы define-method-combination.

(:generic-function-class *symbol*)

Определяет принадлежность обобщенной функции к классу *symbol*. Может использоваться для изменения класса существующей обобщенной функции. По умолчанию используется standard-generic-function.

(:method-class *symbol*)

Указывает, что все методы обобщенной функции должны принадлежать классу *symbol*. Могут переопределяться классы существующих методов. По умолчанию используется standard-method.

(:method *qualifier* parameters . body*)

Эквивалент (defmethod *fname qualifier* parameters . body*). *entry* может включать более одного выражения такого типа.

(define-method-combination *symbol property**) макрос

Сокращенная форма: определяет новый тип комбинации методов. Краткая форма используется для прямой операторной комбинации методов. Если $c_1 \ldots c_n$ – список вызовов применимых методов (от более специфичного к менее специфичному) для комбинации методов с оператором *symbol*, то вызов обобщенной функции будет эквивалентен (*symbol* $c_1 \ldots c_n$). *property* может быть следующим:

:documentation *string*

Задает строку документации для *symbol* с ключом method-combination.

:identity-with-one-argument *bool*

Делает возможной оптимизацию вызовов обобщенной функции, для которой существует только один применимый метод. Если *bool* истинен, значения, возвращаемые методом, возвращаются как значения обобщенной функции. Используется, например, в and- и progn-комбинации методов.

`:operator` *opname*

Определяет актуальный оператор (может отличаться от *symbol*), используемый обобщенной функцией; *opname* может быть символом или лямбда-выражением.

(`define-method-combination` *symbol parameters* (*group-spec**) [(`:arguments` . *parameters2*)] [(`:generic-function` *var*)] . *body*) макрос

Полная форма: определяет новую форму комбинации методов путем определения раскрытия вызова обобщенной функции. Вызов обобщенной функции, использующей *symbol*-комбинацию, будет эквивалентен выражению, возвращаемому телом *body*. При вычислении этого выражения единственной локальной связью будет макроопределение для `call-method`.

Формы, предшествующие *body*, связывают переменные, которые могут использоваться для генерации раскрытия. *parameters* получает список параметров, следующих за *symbol* в аргументе `:method-combination` в `defgeneric`. *parameters2* (если задан) получает форму, которая появляется в вызове обобщенной функции. Оставшиеся необязательные параметры получают соответствующие initform-значения. Кроме того, допустимо использование параметра `&whole`, который получает список всех аргументов формы. *var* (если задан) будет связан с самим объектом обобщенной функции.

Выражения *group-spec* могут использоваться для связывания переменных с непересекающимися списками применимых методов. Каждым *group-spec* может быть форма вида (*var* {*pattern** | *predname*} *option**). Каждая переменная *var* будет связана со списком методов, чьи квалификаторы совпадают с шаблоном *pattern* или удовлетворяют предикату с именем *predname*. (Если не задан ни один предикат, то должен быть представлен по крайней мере один шаблон.) Шаблоном может быть `*`, что соответствует любому списку квалификаторов, или список символов, который совпадает (`equal`) со списком квалификаторов. (Этот список может также содержать `*` как элемент или как хвост списка.) Метод для заданного списка квалификаторов будет аккумулироваться в первой переменной *var*, предикат которой возвратит истину для данного списка или же один из шаблонов которой обеспечит совпадение. Доступны следующие опции:

`:description` *format*

Некоторые утилиты используют *format* как второй аргумент в вызове `format`, третьим аргументом которой является список квалификаторов метода.

:order *order*

Если значение *order* равно :most-specific-first (по умолчанию), методы аккумулируются по убыванию специфичности; если используется :most-specific-last, то в обратном порядке.

:required *bool*

Если значение *bool* истинно, то в случае отсутствия аккумулированных методов будет возвращаться ошибка.

(defmethod *fname qualifier* parameters* . *body*) макрос

Определяет метод обобщенной функции *fname*; если она не существует, то будет создана. Возвращает новый метод. Вызывает ошибку, если *fname* является именем обычной функции или макроса.

Квалификаторы *qualifier* – это атомы, используемые при комбинации методов. Стандартная комбинация методов допускает квалификаторы :before, :after или :around.

Параметры *parameter* подобны аналогичным в обычных функциях, за исключением того, что обязательные параметры могут быть представлены списками вида (*name specialization*), где *specialization* – класс или имя класса (первый вариант) либо список вида (eql *expression*) (второй вариант). В первом случае соответствующий аргумент должен принадлежать указанному классу. Второй вариант указывает, что параметр в момент раскрытия выражения defmethod должен быть равен (eql) *expression*. Методы определяются уникальным образом по квалификаторам и специализации, и если два метода идентичны, то новый заменит старый. Список *parameters* должен быть конгруэнтен параметрам других методов данной обобщенной функции, а также самой обобщенной функции.

Применение метода эквивалентно вызову (lambda *parms* . *body*) с аргументами обобщенной функции, где *parms* – это *parameters* без специализации. Как и для defun, тело неявно заключается в блок с именем *fname*, если *fname* – символ, или *f*, если *fname* – список вида (setf *f*).

(ensure-generic-function *fname* &key *argument-precedence-order* *declare* *documentation* *environment* *generic-function-class* *lambda-list* *method-class* *method-combination*) функция

Делает *fname* (который не может быть именем обычной функции или макроса) именем обобщенной функции с заданными свойствами. Если обобщенная функция с таким именем уже существует, ее свойства заменяются при выполнении следующих условий. Свойства *argument-precedence-order*, *declare*, *documentation* и *method-combination* перезаписываются всегда; *lambda-list* должен быть конгруэнтен

спискам параметров существующих методов; *generic-function-class* должен быть совместим со старым значением, и в этом случае для замены значения вызывается `change-class`. Когда изменяется *method-class*, существующие методы остаются без изменения.

(`find-class` *symbol* `&optional` *error environment*) функция

Возвращает класс, именем которого в *environment* является *symbol*. Если такой класс отсутствует, вызывается ошибка, когда *error* истинна (по умолчанию), иначе возвращается `nil`. Может быть первым аргументом `setf`; чтобы отвязать имя от класса, необходимо установить `find-class` для этого класса в `nil`.

(`find-method` *generic-function qualifiers specializers* `&optional` *error*) обобщенная функция

Возвращает метод обобщенной функции *generic-function*, квалификаторы которой совпадают с *qualifiers*, а специализация – с *specializers*. *specializers* – это список классов (не имен); класс `t` соответствует отсутствию специализации. Если метод не найден, истинность *error* вызывает ошибку, иначе возвращается `nil`.

(`function-keywords` *method*) обобщенная функция

Возвращает два значения: список аргументов по ключу, применимых к методу *method*; второе значение истинно, когда метод *method* допускает использование других ключей.

(`initialize-instance` *instance* `&rest` *initargs* `&key`) обобщенная функция

Встроенный первичный метод вызывает `shared-initialize` для установки слотов *instance* согласно заданным *initargs*. Используется в `make-instance`. Допускает использование других ключей.

(`make-instance` *class* `&rest` *initargs* `&key`) обобщенная функция

Возвращает новый экземпляр класса *class*. Вместо *initargs* должны быть пары символ-значение: $k_1\ v_1 \ldots k_n\ v_n$. Каждый слот в новом экземпляре будет инициализироваться следующим образом: если параметр k является параметром слота, то значением слота будет соответствующий ему v; иначе, если класс или один из его суперклассов имеет *initarg*-параметры по умолчанию, в том числе для данного слота, то будет использоваться значение по умолчанию для наиболее специфичного класса; иначе, если слот имеет `initform`, слот получает ее значение; в противном случае слот остается несвязанным. Допускает использование других ключей.

(`make-instance-obsolete` *class*) обобщенная функция

Вызывается из `defclass` при попытке изменения определения класса *class*. Обновляет все экземпляры класса (вызывая `update-instance-for-redefined-class`) и возвращает *class*.

(make-load-form *object* &optional *environment*) обобщенная функция

Если объект *object* является экземпляром, структурой, исключением или классом, возвращает одно или два выражения, которые при вычислении в *environment* приводят к значению, эквивалентному *object* в момент загрузки.

(make-load-form-saving-slots *instance* &key *slot-names environment*) функция

Возвращает два выражения, которые при вычислении в *environment* приведут к значению, эквивалентному *instance* в момент загрузки. Если заданы *slot-names*, то будут сохранены лишь перечисленные слоты.

(method-qualifiers *method*) обобщенная функция

Возвращает список квалификаторов метода.

(next-method-p) функция

Вызванная внутри метода, возвращает истину в случае наличия следующего метода.

(no-applicable-method *generic-function* &rest *args*) обобщенная функция

Вызывается, когда для обобщенной функции *generic-function* с заданными аргументами *args* нет ни одного применимого метода. Встроенный первичный метод сигнализирует об ошибке.

(no-next-method *generic-function method* &rest *args*) обобщенная функция

Вызывается, когда метод *method* обобщенной функции *generic-function* пытается вызвать следующий метод, который не существует. *args* – аргументы несуществующего следующего метода. Встроенный первичный метод сигнализирует об ошибке.

(reinitialize-instance *instance* &rest *initargs*) обобщенная функция

Устанавливает значения слотов экземпляра *instance* в соответствии с *initargs*. Встроенный первичный метод передает аргументы в shared-initialize (со вторым аргументом nil). Допускает использование других ключей.

(remove-method ⟨*generic-function*⟩ *method*) обобщенная функция

Деструктивно удаляет метод *method* из обобщенной функции *generic-function*, возвращая *generic-function*.

(shared-initialize *instance names* &rest *initargs* &key) обобщенная функция

Устанавливает слоты экземпляра *instance* в соответствии с *initargs*. Все оставшиеся слоты инициализируются согласно их initform, если их имена перечислены в *names* или *names* равен t. Допускает использование других ключей.

(slot-boundp *instance symbol*) функция

Возвращает истину, когда слот с именем *symbol* в заданном экземпляре *instance* инициализирован. Если такого слота нет, вызывает slot-missing.

(slot-exists-p *object symbol*) функция

Возвращает истину, когда объект *object* имеет слот с именем *symbol*.

(slot-makunbound ⟨*instance*⟩ *symbol*) функция

Отвязывает слот с именем *symbol* в заданном экземпляре *instance*.

(slot-missing *class object symbol opname* &optional *value*) обобщенная функция

Вызывается, когда оператор с именем *opname* не смог найти слот с именем *symbol* в объекте *object* класса *class*. (Если задано значение *value*, то оно устанавливается вместо несуществующего значения слота.) Встроенный первичный метод сигнализирует об ошибке.

(slot-unbound *class instance symbol*) обобщенная функция

Вызывается, когда slot-value запрашивает значение слота *symbol* в экземпляре *instance* (чей класс – *class*), а этот слот является несвязанным. Встроенный первичный метод сигнализирует об ошибке. Если новый метод возвращает значение, это значение будет возвращено из slot-value.

(slot-value *instance symbol*) функция

Возвращает значение слота *symbol* в экземпляре *instance*. Если слот отсутствует, вызывает slot-missing, если же слот несвязан – slot-unbound. По умолчанию в обоих случаях вызывается ошибка. Может быть первым аргументом setf.

(with-accessors ((*var fname*)*) *instance declaration* expression**) макрос

Вычисляет свое тело, предварительно связав каждую переменную *var* с результатом вызова соответствующей функции для *instance*. Каждое *fname* должно быть именем функции доступа для экземпляра.

(with-slots ({*symbol* | (*var symbol*)}*) *instance declaration* expression**) макрос

Вычисляет свое тело, предварительно связав каждый символ (или *var*, если задан) с локальным символом-макросом, ссылающимся на слот с именем *symbol* для экземпляра *instance*.

(unbound-slot-instance *condition*) функция

Возвращает экземпляр, чей слот не был связан в момент возникновения исключения *condition*.

(update-instance-for-different-class *old new* &rest *initargs* &key) обобщенная функция

Вызывается из change-class для установления значений слотов при изменении класса экземпляра. Экземпляр *old* является копией оригинального экземпляра в динамическом диапазоне; *new* – оригинальный экземпляр, к которому добавляются требуемые слоты. Встроенный первичный метод использует shared-initialize со следующими аргументами: *new*, список имен новых слотов и *initargs*. Допускает использование других ключей.

(update-instance-for-redefined-class *instance added deleted plist* &rest *initargs*) обобщенная функция

Вызывается из make-instance-obsolete для установления значений слотов при изменении класса экземпляра; *added* – это список добавляемых слотов, *deleted* – список удаляемых (включая те, которые перешли из локальных в разделяемые); *plist* содержит элементы вида (*name* . *val*) для каждого элемента *deleted*, имевшего значение *val*. Встроенный первичный метод вызывает shared-initialize с аргументами: *instance*, *added* и *initargs*.

Структуры

(copy-structure *structure*) функция

Возвращает новую структуру того же типа, что и *structure*, все значения полей которой равны (eql) старым.

(defstruct {*symbol* | (*symbol property**)} [*string*] *field**) макрос

Определяет новый тип структуры с именем *symbol*, возвращая *symbol*. Если *symbol* уже соответствует имени структуры, последствия не определены, но переопределение структуры с теми же параметрами, как правило, безопасно. Если *symbol* равен str, тогда по умолчанию также определяются: функция make-str, возвращающая новую структуру; предикат str-p, проверяющий на принадлежность к str; функция copy-str, копирующая str; функции доступа к каждому полю str; тип с именем str.

Строка *string*, если представлена, становится документацией к *symbol* с ключом structure. По умолчанию она также становится документацией к *symbol* с ключом type, а также документацией для сопутствующего класса.

property может быть одним из:

:conc-name | (:conc-name [*name*])

Функция для доступа к полю f структуры str будет называться *namef* вместо str-f (по умолчанию). Если *name* не указано или является nil, именем функции будет просто f.

`:constructor` | (`:constructor` [*name* [*parameters*]])

Если *name* указано и не равно `nil`, функция создания новых структур будет называться *name*. Если *name* равно `nil`, такая функция не будет определена. Если имя не задается вовсе, будет использоваться имя по умолчанию, `make-str`. Если задан список полей *parameters*, он становится списком параметров функции-конструктора, и каждое поле вновь создаваемой структуры будет иметь значение, переданное соответствующим аргументом. Для одной структуры могут быть определены несколько конструкторов.

`:copier` | (`:copier` [*name*])

Если *name* задано и не является `nil`, функция копирования структуры будет называться *name*. Если *name* является `nil`, такая функция не будет определена. Если имя не задается вовсе, используется имя по умолчанию, `copy-str`.

(`:include` *name field**)

Означает, что все структуры `str` будут также включать все поля существующей структуры типа с именем *name*. Функции доступа включаются вместе с полями. Поля *fields* в выражении `:include` используют обычный синтаксис (см. ниже); они могут использоваться для задания исходного значения поля (или его отсутствия), или для установки флага «только чтение», или для спецификации типа поля (который должен быть подтипом оригинального). Если структура содержит параметр `:type` (см. ниже), включаемая структура должна быть того же типа, в противном случае тип включаемой структуры будет подтипом новой.

(`:initial-offset` *i*)

Структуры будут размещаться со смещением в *i* неиспользуемых полей. Используется только вместе с `:type`.

`:named`

Структуры будут размещаться так, что первым элементом является имя структуры. Используется только вместе с `:type`.

`:predicate` | (`:predicate` [*name*])

Если *name* задано и не является `nil`, предикат идентификации структуры будет называться *name*. Если *name* является `nil`, такая функция не будет определена. Если имя не задается вовсе, будет использовано имя по умолчанию, `str-p`. Не может использоваться вместе с `:type` до тех пор, пока не задано `:named`.

(`:print-function` [*fname*])

Когда `str` выводится на печать, функция с именем *fname* (которая также может быть лямбда-выражением) будет вызываться с тремя аргументами: структура, поток, в который выполняется печать, и целое число, представляющее глубину печати. Реализована добавлением метода `print-object`. Не может быть использована вместе с `:type`.

(:print-object [*fname*])

Действует подобно :print-function, но вызывается лишь с двумя аргументами. Может использоваться что-то одно: либо :print-function, либо :print-object.

(:type {vector | (vector *type*) | list})

Указывает способ реализации структуры в виде объекта заданного типа. Отдельные структуры в таком случае будут списками или векторами; никакой новый тип не будет определен для структуры, и никакой предикат для идентификации структур (если только не задано :named) не будет добавлен. Если заданный тип :type – (vector *type*), то :named может использоваться, только если *type* является надтипом symbol.

Каждое поле *field* может быть одиночным символом *name* или (*name* [*initform property**]).

Имя одного поля не должно совпадать с именем другого, включая локальные и наследуемые через :include. Имя поля будет использоваться при создании функции доступа к этому полю; по умолчанию для структуры str будет определяться функция str-name, но это поведение можно настроить с помощью :conc-name. Имя *name* также становится параметром по ключу в функции создания структур str, значение которого будет помещено в соответствующее поле новой структуры.

Исходное значение *initform*, если задано, вычисляется каждый раз при создании новой структуры в том окружении, где произошел вызов defstruct. Если исходное значение *initform* не задано, значение поля на момент создания не определено. *property* может быть одним из:

:type *type*

Декларирует, что данное поле будет содержать объекты только типа *type*.

:read-only[1] *bool*

Если *bool* не является nil, поле будет «только для чтения».

Особые условия

(abort &optional *condition*) функция

Вызывает рестарт, возвращаемый (find-restart 'abort *condition*).

(assert *test* [(*place**) [*cond arg**]]) макрос

Если вычисление *test* возвращает nil, вызывается корректируемая ошибка, заданная значениями *cond* и *args*. Значение *test* должно зависеть от мест *places*; имеется возможность присвоить им новые зна-

[1] Исправлена авторская опечатка, обнаруженная Лорентом Пероном. – *Прим. перев.*

чения и продолжить вычисление. При корректном завершении возвращает nil.

(break &rest *args*) функция

Вызывает format с аргументами *args*, затем вызывает отладчик. Не сигнализирует об особом условии.

(cell-error-name *condition*) функция

Возвращает имя объекта, вызвавшего особое условие, которое находится в ячейке error объекта *condition*.

(cerror *format cond* &rest *args*) функция

Действует подобно error, но позволяет продолжить вычисление с момента возникновения ошибки, возвращая при этом nil. Строка *format* передается format при отображении информации об ошибке.

(check-type *place type* [*string*]) макрос

Вызывает корректируемую ошибку, если значение места *place* не принадлежит типу *type*. Если задана строка *string*, то она будет выводиться как описание требуемого типа.

(compute-restarts &optional *condition*) функция

Возвращает список ожидающих рестартов, от последнего к первому. Если передано условие *condition*, список будет содержать рестарты, связанные с данным условием или не связанные ни с каким условием, в противном случае он будет содержать все рестарты. Возвращаемый список нельзя модифицировать.

(continue &optional *condition*) функция

Если функция (find-restart 'abort *condition*) находит рестарт, она вызывает его, иначе возвращает nil.

(define-condition *name* (*parent**) (*slot-spec**) *class-spec**) макрос

Определяет новый тип условий и возвращает его имя. Имеет такой же синтаксис и поведение, как и defclass, за исключением того, что *class-specs* не может включать предложение :metaclass, но может включать :report. Предложение :report определяет способ вывода отчета об условии. Этот аргумент может быть символом или лямбда-выражением, соответствующим функции двух аргументов (условия и потока), либо может быть строкой.

(error *cond* &rest *args*) функция

Сигнализирует о простой ошибке для *cond* и *args*. Если она не перехватывается обработчиком, вызывается отладчик.

(find-restart *r* &optional *condition*) функция

Возвращает самый недавний ожидающий рестарт с именем *r*, если *r* – символ, или с именем, равным (eq) *r*, если *r* – рестарт. Если передано условие *condition*, учитываются лишь рестарты, связанные

с ним, или не связанные ни с каким условием. Если заданный рестарт не найден, возвращает nil.

(handler-bind ((*type handler*)*) *expression**) макрос

Вычисляет выражения *expression* с локальными обработчиками условий. Если сигнализируется условие, оно передается в функцию (в качестве одного из аргументов), связанную с первым *handler*, чей тип *type* соответствует условию. Если обработчик отказывается от обработки условия (возвращая управление), поиск продолжается. Пройдя таким образом все локальные обработчики, система начинает искать обработчики, существовавшие на момент вычисления выражения handler-bind.

(handler-case *test* *макрос*
(*type* ([*var*]) *declaration** *expression**)*
[(:no-error *parameters declaration** *expression**)])

Вычисляет *test*. Если обнаруживается условие и оно принадлежит одному из типов *type*, тогда оно обрабатывается, и выражение handler-case возвращает результат(ы) вычисления выражений *expressions*, связанных с первым совпавшим типом, при этом var (если задано) связывается с условием. Если условие не обнаружено и не задано поведение :no-error, тогда выражение handler-case возвращает результат выражения *test*. Если задано :no-error, handler-case возвращает результат(ы) вычисления его выражений, связав *parameters* со значениями, возвращенными *test*. Предложение :no-error может идти как первым, так и последним.

(ignore-errors *expression**) макрос

Действует подобно progn, за исключением того, что выражения *expression* вычисляются с локальным обработчиком всех ошибок. Этот обработчик приводит к тому, что возвращается два значения: nil и вызванное особое условие.

(invalid-method-error *method format* &rest *args*) функция

Используется для сигнализации ошибки, когда один из применимых методов содержит некорректный квалификатор. Параметры *format* и *arg* передаются format для отображения сообщения об ошибке.

(invoke-debugger *condition*) функция

Вызывает отладчик для заданного особого условия *condition*.

(invoke-restart *restart* &rest *args*) функция

Если *restart* – рестарт, вызывает его функцию с аргументами *args*; если это символ, вызывает функцию самого недавнего ожидающего рестарта с этим именем и с аргументами *args*.

(invoke-restart-interactively *restart*) функция

Действует подобно invoke-restart, но предоставляет интерактивную подсказку для ввода аргументов.

(make-condition *type* &rest *initargs*) функция

Возвращает новое условие типа *type*. По сути, это специализированная версия make-instance.

(method-combination-error *format* &rest *args*) функция

Используется для сигнализации об ошибке при комбинации методов. Аргументы передаются format для отображения сообщения об ошибке.

(muffle-warning &optional *condition*) функция

Вызывает рестарт, возвращаемый (find-restart 'muffle-warning *condition*).

(restart-bind ((*symbol function* {*key val*}*)*) *expression**) макрос

Вычисляет выражения с новым ожидающим рестартом. Каждый символ *symbol* становится именем рестарта с соответствующей функцией *function*. (Если символ *symbol* является nil, рестарт будет безымянным.) Ключи *key* могут быть следующими:

:interactive-function

Соответствующее значение *val* должно вычисляться в функцию без аргументов, строящую список аргументов для invoke-restart. По умолчанию не передаются никакие аргументы.

:report-function

Соответствующее значение *val* должно вычисляться в функцию одного аргумента (поток), печатающую в поток описание действий рестарта.

:test-function

Соответствующее значение *val* должно вычисляться в функцию одного аргумента (условие), которая возвращает истину, если рестарт применим с данным условием. По умолчанию рестарт применим с любым условием.

(restart-case *test* (*symbol parameters* {*key val*}* *declaration** *expression**)*) макрос

Вычисляет *test* с новыми ожидающими рестартами. Каждый символ *symbol* становится именем рестарта с функцией (lambda *parameters declaration* expression**). (Если символ *symbol* является nil, рестарт будет безымянным.) Ключи *key* могут быть следующими:

:interactive

Соответствующее значение *val* должно быть символом или лямбда-выражением, описывающим функцию без аргументов, строящую список аргументов для invoke-restart. По умолчанию не передаются никакие аргументы.

`:report`

Соответствующее значение *val* может быть строкой, описывающей действия рестарта, или же символом или лямбда-выражением, описывающим функцию одного аргумента (поток), выводящую в поток описание рестарта.

`:test`

Соответствующее значение *val* должно быть символом или лямбда-выражением, описывающим функцию одного аргумента (условие), которая истинна, когда рестарт применим с данным условием. По умолчанию рестарт применим с любым условием.

(`restart-name` ***restart***) функция

Возвращает имя рестарта *restart* или `nil`, если он безымянный.

(`signal` ***cond*** `&rest` ***args***) функция

Сигнализирует об условии, задаваемым *cond* и *args*. Если оно не обрабатывается, возвращает `nil`.

(`simple-condition-format-arguments` ***condition***) функция

Возвращает параметры `format` для `simple-condition`.

(`simple-condition-format-control` ***condition***) функция

Возвращает строку (или функцию) `format` для `simple-condition`.

(`store-value` ***object*** `&optional` ***condition***) функция

Вызывает рестарт, возвращаемый (`find-restart 'store-value` ***condition***) для объекта *object*. Если рестарт не найден, возвращает `nil`.

(`use-value` ***object*** `&optional` ***condition***) функция

Вызывает рестарт, возвращаемый (`find-restart 'use-value` ***condition***) для объекта *object*. Если рестарт не найден, возвращает `nil`.

(`warn` ***cond*** `&rest` ***args***) функция

Сигнализирует о предупреждении `simple-warning`, задаваемым *cond* и *args*. Если оно не обрабатывается, то предупреждение печатается в `*error-output*` и возвращается `nil`.

(`with-condition-restarts` ***condition restarts expression****) макрос

Сначала вычисляется условие *condition*, затем для создания списка рестартов вычисляется *restarts*. После этого все рестарты связываются с полученным условием и вычисляются выражения *expression*.

(`with-simple-restart` (***symbol format arg****) ***expression****) макрос

Вычисляет выражения e*xpression* с новым рестартом с именем *symbol*, который, будучи вызванным, заставляет выражение `with-simple-restart` возвращать два значения: `nil` и `t`. Аргументы *format* и *args* передаются в `format` для вывода описания рестарта.

Символы

(boundp *symbol*) функция

Возвращает истину, если символ *symbol* является именем специальной переменной.

(copy-symbol *symbol* &optional *props-too*) функция

Возвращает новый неинтернированный символ с именем, равным (string=) данному *symbol*. Если параметр *props-too* истинен, новый символ будет иметь те же symbol-value и symbol-function, что и *symbol*, а также копию его symbol-plist.

(gensym &optional *prefix*) функция

Возвращает новый неинтернированный символ. По умолчанию его имя начинается с "G" и содержит увеличенное представление значения счетчика *gensym-counter*. Если предоставлена строка-префикс *prefix*, то она будет использоваться вместо "G".

(gentemp &optional (*prefix* "T") *package*) [функция]

Возвращает новый символ, интернированный в пакет *package*. Имя символа содержит префикс *prefix* и значение внутреннего счетчика, которое увеличивается, если символ с таким именем уже существует.

(get *symbol key* &optional *default*) функция

Если список свойств символа *symbol* – (k_1 v_1...k_n v_n) и *key* равен (eq) некоторому *k*, возвращает значение *v* для заданного ключа *k*. Возвращает *default*, если искомое свойство отсутствует (нет такого *k*). Может быть первым аргументом setf. *k*

(keywordp *object*) функция

Возвращает истину, когда *object* является символом из пакета keyword.

(make-symbol *string*) функция

Возвращает новый неинтернированный символ с именем, равным (string=) строке *string*.

(makunbound *symbol*) функция

Удаляет специальную переменную с именем *symbol*, если такая имеется. После этого (boundp *symbol*) больше не будет возвращать истину. Возвращает *symbol*.

(set *symbol object*) [функция]

Эквивалент (setf (symbol-value *symbol*) *object*).

(symbol-function *symbol*) функция

Возвращает глобальную функцию с именем *symbol*. Сигнализирует об ошибке, если функция с таким именем отсутствует. Может быть первым аргументом setf.

(symbol-name *symbol*) функция

Возвращает строку, являющуюся именем символа *symbol*. Строка не может быть модифицирована.

(symbolp *object*) функция

Возвращает истину, когда объект *object* является символом.

(symbol-package *symbol*) функция

Возвращает домашний пакет для символа *symbol*.

(symbol-plist *symbol*) функция

Возвращает список свойств символа *symbol*. Может быть первым аргументом setf.

(symbol-value *symbol*) функция

Возвращает значение специальной переменной с именем *symbol*. Сигнализирует об ошибке, если такая переменная не существует. Может быть первым аргументом setf.

(remprop ⟨*symbol*⟩ *key*) функция

Деструктивно удаляет первый найденный в списке свойств символа *symbol* ключ *key* и связанное с ним значение. Возвращает истину, если заданный ключ *key* был найден.

Пакеты

(defpackage *name property**) макрос

Возвращает пакет с именем *name* (или его имя, если это символ) и заданными свойствами. Если пакета с таким именем не существует, он создается; иначе вносятся изменения в уже имеющийся пакет. Свойствами *property* могут быть:

(:nicknames *name**)

Делает имена *name* (символы или строки) альтернативными именами пакета.

(:documentation *string*)

Делает *string* строкой документации к пакету.

(:use *package**)

Определяет пакеты, используемые данным *package*; см. use-package.

(:shadow *name**)

Имена *name* могут быть символами или строками; соответствующие символы будут определять затененные в пакете символы; см. shadow.

(:shadowing-import-from *package name**)

Имена *name* могут быть символами или строками; соответствующие символы из *package* будут импортироваться в пакет так же, как с помощью shadowing-import.

(:import-from *package name**)

Имена *name* могут быть символами или строками; соответствующие символы из *package* будут импортироваться в пакет так же, как с помощью import.

(:export *name**)

Имена *name* могут быть символами или строками; соответствующие символы будут экспортироваться из *package*; см. export.

(:intern *name**)

Имена *name* могут быть символами или строками; соответствующие символы создаются в пакете, если они еще не существуют; см. intern.

(:size *integer*)

Декларирует ожидаемое количество символов в данном пакете.

Любые свойства *property*, кроме :documentation и :size, могут встречаться неоднократно. Свойства применяются в следующем порядке: :shadow и :shadowing-import-from, затем :use, затем :import-from и :intern, затем :export. Вся работа выполняется на этапе компиляции, если выражение находится в toplevel.

(delete-package *package*) функция

Удаляет пакет *package* из списка активных, но при этом представляющий его объект не затрагивается. Возвращает истину, если до удаления пакет *package* находился в списке активных.

(do-all-symbols (*var* [*result*]) макрос
*declaration** {*tag* | *expression*}*)

Действует подобно do-symbols, но итерирует по всем активным пакетам.

(do-external-symbols (*var* [*package* [*result*]]) макрос
*declaration** *tag* | *expression*}*)

Действует подобно do-symbols, но итерирует только по внешним символам пакета *package*.

(do-symbols (*var* [*package* [*result*]]) макрос
*declaration** {*tag* | *expression*}*)

Вычисляет свое тело, связывая *var* по очереди со всеми символами, доступными в пакете *package*. Символы, наследуемые из других пакетов, могут обрабатываться несколько раз. Тело заключается в tagbody, а весь вызов do-symbols – в блок с именем nil. Возвращает значение(я)

выражения *result* или `nil`, если *result* не предоставляется. Выражение *result* может ссылаться на *var*, которая будет `nil`.

(`export` *symbols* `&optional` *package*) функция

Делает каждый символ, доступный в данном пакете (если *symbols* – список, то каждый символ в нем), внешним для пакета *package*. Возвращает `t`.

(`find-all-symbols` *name*) функция

Возвращает список, содержащий все символы активного пакета с именем *name* (если *name* – строка) или имя *name* (если *name* – символ).

(`find-package` *package*) функция

Возвращает пакет, на который указывает *package*, или `nil`, если такого пакета нет.

(`find-symbol` *string* `&optional` *package*) функция

Возвращает символ с именем *string*, доступный в пакете *package*. Второе возвращаемое значение указывает, является ли символ: `:internal`, `:external` или `:inherited`. Если искомый символ с именем *string* не найден, оба возвращаемые значения – `nil`.

(`import` *symbols* `&optional` *package*) функция

Делает каждый из *symbols* (должен быть символом или списком символов; если *symbols* – список, то каждый символ в нем), доступный в данном пакете, доступным в пакете *package*. Для символов, не имеющих домашнего пакета, им становится *package*. Возвращает `t`.

(`in-package` *name*) макрос

Устанавливает текущим пакет, соответствующий *name* (строка или символ). Если выражение расположено в toplevel, работа выполняется на этапе компиляции.

(`intern` *string* `&optional` *package*) функция

Возвращает символ, доступный в *package*, с именем, равным (`string=`) *string*, при необходимости создавая такой символ. Второе возвращаемое значение указывает, является ли символ: `:internal`, `:external`, `:inherited` или `nil` (что означает, что символ был только что создан).

(`list-all-packages`) функция

Возвращает новый список всех активных пакетов.

(`make-package` *name* `&key` *nicknames use*) функция

Возвращает новый пакет с именем *name* (или его имя, если *name* – символ), альтернативными именами *nicknames* (это строки в списке *nicknames* и имена любых символов в нем) и используемыми пакетами, указанными в *use* (список пакетов и/или указывающие на них строки и символы).

(package-error-package *condition*) функция

Возвращает пакет, вызвавший условие *condition*, связанное с ошибкой обращения к пакету.

(package-name *package*) функция

Возвращает строку, являющуюся именем пакета *package*, или nil, если пакет не является активным.

(package-nicknames *package*) функция

Возвращает список строк, являющихся альтернативными именами пакета *package*.

(packagep *object*) функция

Возвращает истину, если объект *object* является пакетом.

(package-shadowing-symbols *package*) функция

Возвращает список затененных символов для пакета *package*.

(package-used-by-list *package*) функция

Возвращает список пакетов, в которых используется пакет *package*.

(package-use-list *package*) функция

Возвращает список пакетов, используемых пакетом *package*.

(rename-package ⟨*package*⟩ *name* &optional *nicknames*) функция

Дает пакету *package* новое имя *name* (если это строка) или имя символа *name* (если это символ), а также устанавливает новые альтернативные имена – строки из списка *nicknames* и имена присутствующих в нем символов. Возвращает пакет.

(shadow *names* &optional *package*) функция

names может быть строкой, символом, списком строк и/или символов. Добавляет указанные символы в список затеняемых для данного пакета *package*. Несуществующие символы при необходимости создаются в *package*. Возвращает t.

(shadowing-import *symbols* &optional *package*) функция

Делает каждый из *symbols* (должен быть символом или списком символов; если *symbols* – список, то каждый символ в нем) внутренним для *package* и добавляет его в список затененных. Если в пакете уже существует доступный символ с таким именем, он удаляется. Возвращает t.

(unexport *symbols* &optional *package*) функция

Делает символ (если *symbols* – список, то каждый символ этого списка) внутренним для *package*. Возвращает t.

(unintern *symbol* &optional *package*) функция

Удаляет символ *symbol* из пакета *package* (и из списка затененных символов). Если пакет *package* является домашним для символа *symbol*, символ удаляется совсем. Возвращает истину, если символ *symbol* был доступен в пакете *package*.

(unuse-package *packages* &optional *package*) функция

Отменяет эффект use-package, используя те же аргументы. Возвращает t.

(use-package *packages* &optional *package*) функция

Делает все внешние символы пакетов *packages* (это может быть пакет, строка, символ или список из них) доступными в *package*. Не может использоваться пакет keyword. Возвращает t.

(with-package-iterator (*symbol packages key**) *declaration** *expression**) макрос

Вычисляет выражения *expression*, связав *symbol* с локальным макросом, последовательно возвращающим символы из пакетов *packages* (это может быть пакет, символ, строка или список из них). Ключи :internal, :external и :inherited могут ограничивать круг символов. Локальный макрос возвращает четыре значения: первое значение истинно, если возвращается символ (при этом nil означает холостой запуск); второе значение – символ; третье – ключ, характеризующий символ (:internal, :external или :inherited); четвертое – пакет, из которого был получен символ. Локальный макрос может возвращать символы в любом порядке, а также может возвращать один символ несколько раз, если он наследуется из нескольких пакетов.

Числа

(abs *n*) функция

Возвращает неотрицательное действительное число той же величины, что и *n*.

(acos *n*) функция

Возвращает арккосинус *n* (в радианах).

(acosh *n*) функция

Возвращает гиперболический арккосинус *n*.

(arithmetic-error-operands *condition*) функция

Возвращает список операндов арифметической ошибки *condition*.

(arithmetic-error-operation *condition*) функция

Возвращает оператор (или его имя) арифметической ошибки *condition*.

(ash *i pos*) функция

Возвращает целое число, полученное сдвигом числа в дополнительном коде[1] *i* на *pos* позиций влево (или вправо, если *pos* меньше нуля).

(asin *n*) функция

Возвращает арксинус *n* (в радианах).

(asinh *n*) функция

Возвращает гиперболический арксинус *n*.

(atan *n1* &optional (*n2* 1)) функция

Возвращает арктангенс *n1*/*n2* (в радианах).

(atanh *n*) функция

Возвращает гиперболический арктангенс *n*.

(boole *op i1 i2*) функция

Возвращает целое число, полученное применением логического оператора *op* к числам в дополнительном коде *i1* и *i2*. Common Lisp определяет 16 констант, представляющих побитовые логические операции. В следующей таблице приведены значения, возвращаемые boole для различных операторов:

Оператор	Результат
boole-1	*i1*
boole-2	*i2*
boole-andc1	(logandc1 *i1 i2*)
boole-andc2	(logandc2 *i1 i2*)
boole-and	(logand *i1 i2*)
boole-c1	(lognot *i1*)
boole-c2	(lognot *i2*)
boole-clr	всегда 0
boole-eqv	(logeqv *i1 i2*)
boole-ior	(logior *i1 i2*)
boole-nand	(lognand *i1 i2*)
boole-nor	(lognor *i1 i2*)
boole-orc1	(logorc1 *i1 i2*)
boole-orc2	(logorc2 *i1 i2*)
boole-set	всегда 1
boole-xor	(logxor *i1 i2*)

[1] Дополнительный код (two's complement) – распространенное машинное представление отрицательных чисел. В такой записи старший разряд определяет знак числа: 0 – для положительных, 1 – для отрицательных чисел. Далее такое представление для краткости может называться «дополнительным». – *Прим. перев.*

(byte *length pos*) функция

Возвращает спецификатор байта длиной *length* бит, нижний бит которого представляет 2^{pos}.

(byte-position *spec*) функция

Возвращает $\log_2$ числа, представляющего значение нижнего бита для байт-спецификатора *spec*.

(byte-size *spec*) функция

Возвращает количество бит для байт-спецификатора *spec*.

(ceiling *r* &optional (*d* 1)) функция

Возвращает два значения: наименьшее целое число i, большее или равное r/d, и $r - i \times d$. Число d должно быть ненулевым действительным числом.

(cis *r*) функция

Возвращает комплексное число с действительной частью (cos *r*) и мнимой частью (sin *r*).

(complex *r1* &optional *r2*) функция

Возвращает комплексное число с действительной частью *r1* и мнимой частью *r2* или ноль, если *r2* не задано.

(complexp *object*) функция

Возвращает истину, когда объект *object* является комплексным числом.

(conjugate *n*) функция

Возвращает результат комплексного сопряжения n: n, если n действительное число, и #c(a $-b$), если n равно #c(a b).

(cos *n*) функция

Возвращает косинус n (в радианах).

(cosh *n*) функция

Возвращает гиперболический косинус n.

(decf *place* [*n*]) макрос

Уменьшает значение места *place* на n или на 1, если n не задано.

(decode-float *f*) функция

Возвращает три значения: мантиссу f, его экспоненту и знак (-1.0 для отрицательного, в противном случае 1.0). Первое и третье значения – числа с плавающей запятой того же типа, что и f, а второе – целое число.

(denominator *rational*) функция

Если *rational* – правильная рациональная дробь a/b, возвращает b.

(deposit-field *new spec i*) функция

Возвращает результат замены битов в числе *i* согласно байт-спецификатору *spec* на соответствующие биты *new*.

(dpb *new spec i*) функция

Возвращает результат замены младших *s* бит на *new* в числе *i* согласно байт-спецификатору *spec*, определяющему размер *s*.

(evenp *i*) функция

Возвращает истину, если *i* четное.

(exp *n*) функция

Возвращает e^n.

(expt *n1 n2*) функция

Возвращает $n1^{n2}$.

(fceiling *r* &optional (*d* 1)) функция

Действует подобно ceiling, но первое число возвращается в формате с плавающей запятой.

(ffloor *r* &optional (*d* 1)) функция

Действует подобно floor, но первое число возвращается в формате с плавающей запятой.

(float *n* &optional *f*) функция

Возвращает аппроксимацию с плавающей запятой *n* в формате *f* или single-float, если *f* не задан.

(float-digits *f*) функция

Возвращает количество цифр во внутреннем представлении *f*.

(floatp *object*) функция

Возвращает истину, если объект *object* является числом с плавающей запятой.

(float-precision *f*) функция

Возвращает количество значащих цифр во внутреннем представлении *f*.

(float-radix *f*) функция

Возвращает основание представления *f*.

(float-sign *f1* &optional (*f2* (float 1 *f1*))) функция

Возвращает положительное или отрицательное значение *f2* в зависимости от знака *f1*.

(floor *r* &optional (*d* 1)) функция

Возвращает два значения: наибольшее целое число *i*, меньшее или равное *r*/*d*, и *r*- *i*×*d*. Число *d* должно быть ненулевым действительным числом.

(fround *r* &optional (*d* 1)) функция

Действует подобно round, но первое значение возвращается в формате с плавающей запятой.

(ftruncate *r* &optional (*d* 1)) функция

Действует подобно truncate, но первое значение возвращается в формате с плавающей запятой.

(gcd &rest *is*) функция

Возвращает наибольший общий делитель своих аргументов или 0, если аргументы не заданы.

(imagpart *n*) функция

Возвращает мнимую часть *n*.

(incf *place* [*n*]) макрос

Увеличивает значение места *place* на *n* или на 1, если *n* не задано.

(integer-decode-float *f*) функция

Возвращает три целых числа, связанных друг с другом так же, как значения, возвращаемые decode-float.

(integer-length *i*) функция

Возвращает количество бит, необходимых для представления *i* в виде дополнительного кода.

(integerp *object*) функция

Возвращает истину, если объект *object* является целым числом.

(isqrt *i*) функция

Возвращает наибольшее целое число, меньшее или равное квадратному корню *i* (*i* должно быть положительным числом).

(lcm &rest *is*) функция

Возвращает наименьшее общее кратное своих аргументов или 1, если аргументы не заданы.

(ldb *spec i*) функция

Возвращает целое число, соответствующее битовому представлению *i* в соответствии с байт-спецификатором *spec*. Может быть первым аргументом setf.

(ldb-test *spec i*) функция

Возвращает истину, если любой из битов *i* в соответствии с байт-спецификатором *spec* равен 1.

(log *n1* &optional *n2*) функция

Возвращает $\log_{n2} n1$ или $\log_e n1$, если *n2* не задано.

(logand &rest *is*) функция

Возвращает целочисленный результат логического И дополнительного представления аргументов или 0, если аргументы не заданы.

(logandc1 *i1* *i2*) функция

Возвращает целочисленный результат логического И дополнительных представлений *i2* и дополнения для *i1*.

(logandc2 *i1* *i2*) функция

Возвращает целочисленный результат логического И дополнительных представлений *i1* и дополнения для *i2*.

(logbitp *pos* *i*) функция

Возвращает истину, когда бит с индексом *pos* дополнительного представления *i* равен 1. Индекс младшего бита считается равным 0.

(logcount *i*) функция

Возвращает количество нулей в дополнительном представлении *i*, если *i* отрицательно, иначе – количество единиц.

(logeqv &rest *is*) функция

Возвращает целочисленный результат обратного ИСКЛЮЧАЮЩЕГО ИЛИ[1] для дополнительных представлений аргументов или -1, если аргументы не заданы.

(logior &rest *is*) функция

Возвращает целочисленный результат операции ИСКЛЮЧАЮЩЕГО ИЛИ для дополнительных представлений аргументов или 0, если аргументы не заданы.

(lognand *i1* *i2*) функция

Возвращает целочисленное дополнение результата применения логического И дополнительных представлений аргументов.

(lognor *i1* *i2*) функция

Возвращает целочисленное дополнение результата применения логического ИЛИ дополнительных представлений аргументов.

(lognot *i*) функция

Возвращает целое число, чье дополнительное представление является дополнением *i*.

---

[1] XNOR, логическая операция, возвращающая истину, когда ее аргументы равны. – *Прим. перев.*

(logorc1 *i1 i2*) функция

Возвращает целочисленный результат применения логического ИЛИ для дополнительных представлений *i2* и дополнения до *i1*.

(logorc2 *i1 i2*) функция

Возвращает целочисленный результат применения логического ИЛИ для дополнительных представлений *i1* и дополнения до *i2*.

(logtest *i1 i2*) функция

Возвращает истину, если хотя бы одна из единиц в дополнительном представлении *i1* имеется в дополнительном представлении *i2*.

(logxor &rest *is*) функция

Возвращает целочисленный результат применения логического ИСКЛЮЧАЮЩЕГО ИЛИ к дополнительным представлениям аргументов, или 0, если аргументы не заданы.

(make-random-state &optional *state*) функция

Возвращает новый генератор случайных чисел. Если *state* уже является генератором, возвращается его копия; если nil, то возвращается копия *random-state*; если t, то возвращается генератор, инициализированный случайным образом.

(mask-field *spec i*) функция

Возвращает целое число, представление которого имеет такие же биты, как число *i* в области, определенной байт-спецификатором *spec*.

(max *r1* &rest *rs*) функция

Возвращает наибольший из аргументов.

(min *r1* &rest *rs*) функция

Возвращает наименьший из аргументов.

(minusp *r*) функция

Возвращает истину, если *r* меньше нуля.

(mod *r1 r2*) функция

Возвращает второе значение, которое вернула бы floor с теми же аргументами.

(numberp *object*) функция

Возвращает истину, если объект *object* является числом.

(numerator *rational*) функция

Если *rational* – правильная рациональная дробь вида a/b, возвращает a.

(oddp *i*) функция

Возвращает истину, если *i* нечетное число.

(parse-integer *string* &key *start end radix junk-allowed*) функция

Возвращает два значения: целое число, прочитанное из строки (по умолчанию используется основание 10), и позицию первого непрочитанного знака в ней. Параметры *start* и *end* ограничивают область чтения строки. Строка *string* может содержать пробелы, знаки + и - и должна содержать последовательность цифр, которая при необходимости завершается пробелами. (Макросы чтения не допускаются.) Если параметр *junk-allowed* ложен (по умолчанию), чтение строки *string*, содержащей иные символы, приведет к возникновению ошибки; если он истинен, parse-integer просто вернет nil, если встретит недопустимый символ.

(phase *n*) функция

Возвращает угловую часть числа *n*, представленного в полярных координатах.

(plusp *r*) функция

Возвращает истину, если *r* больше нуля.

(random *limit* &optional (*state* *random-state*)) функция

Возвращает случайное число, меньшее *limit* (который должен быть положительным целым числом или десятичной дробью) и принадлежащее тому же типу. Используется генератор случайных чисел *state* (который изменяется в результате вызова).

(random-state-p *object*) функция

Возвращает истину, когда объект *object* является генератором случайных чисел.

(rational *r*) функция

Преобразует *r* в рациональную дробь. Если *r* является десятичной дробью, преобразование является точным.

(rationalize *r*) функция

Преобразует *r* в рациональную дробь. Если *r* является десятичной дробью, преобразование производится с сохранением точности представленного числа.

(rationalp *object*) функция

Возвращает истину, если объект *object* является рациональной дробью.

(realp *object*) функция

Возвращает истину, если объект *object* является действительным числом.

(realpart *n*) функция

Возвращает действительную часть *n*.

(rem *r1 r2*) функция

Возвращает второе значение вызова truncate с теми же аргументами.

(round *r* &optional (*d* 1)) функция

Возвращает два значения: целое число, ближайшее к r/d, и $r - i \times d$. Если r/d равноудалено от двух целых чисел, выбирается четное. Число d должно быть ненулевым действительным.

(scale-float *f i*) функция

Возвращает результат умножения f на r^i, где r – основание представления числа с плавающей запятой.

(signum *n*) функция

Если n – действительное число, возвращает 1 для положительных чисел, 0 – для нулевых чисел, -1 – для отрицательных чисел. Если n – комплексное число, возвращается комплексное число с величиной, равной 1, и той же фазой.

(sin *n*) функция

Возвращает синус n (в радианах).

(sinh *n*) функция

Возвращает гиперболический синус n.

(sqrt *n*) функция

Возвращает квадратный корень n.

(tan *n*) функция

Возвращает тангенс n (в радианах).

(tanh *n*) функция

Возвращает гиперболический тангенс n.

(truncate *r* &optional (*d* 1)) функция

Возвращает два значения: целое число i, которое получается удалением всех цифр после запятой в десятичном представлении r/d, и $r - i \times d$. Число d должно быть ненулевым действительным числом.

(upgraded-complex-part-type *type*) функция

Возвращает тип частей наиболее специализированного комплексного числа, которое может хранить части с типами *type*.

(zerop *n*) функция

Возвращает истину, если n равен нулю.

(= *n1* &rest *ns*) функция

Возвращает истину, если все аргументы равны попарно.

(/= *n1* &rest *ns*) функция

Возвращает истину, если среди всех аргументов нет двух равных.

(> *r1* &rest *rs*) функция

Возвращает истину, если каждый последующий аргумент меньше предыдущего.

(< *r1* &rest *rs*) функция

Возвращает истину, если каждый последующий аргумент больше предыдущего.

(>= *r1* &rest *rs*) функция

Возвращает истину, если каждый последующий аргумент меньше или равен предыдущему.

(<= *r1* &rest *rs*) функция

Возвращает истину, если каждый последующий аргумент больше или равен предыдущему.

(* &rest *ns*) функция

Возвращает произведение двух аргументов или 1, если аргументы не заданы.

(+ &rest *ns*) функция

Возвращает сумму аргументов или 0, если аргументы не заданы.

(- *n1* &rest *ns*) функция

Вызванная с одним аргументом, возвращает -*n1*. Вызов вида (- a_1... a_n) возвращает a_1 - ... - a_n.

(/ *n1* &rest *ns*) функция

Вызванная с одним аргументом (который не должен быть нулем), возвращает обратный ему. Вызванная с несколькими аргументами, возвращает значение первого, разделенное на произведение остальных (которые не должны включать ноль.)

(1+ *n*) функция

Эквивалент (+ *n* 1).

(1- *n*) функция

Эквивалент (- *n* 1).

Знаки

(alpha-char-p *char*) функция

Возвращает истину, если *char* является буквой.

(both-case-p *char*) функция

Возвращает истину, если *char* имеет регистр.

(alphanumericp *char*) функция

Возвращает истину, если *char* является буквой или цифрой.

(character *c*) функция

Возвращает знак, соответствующий знаку, строке из ровно одного знака или символу, имеющему такую строку в качестве имени.

(characterp *object*) функция

Возвращает истину, если объект *object* является знаком.

(char-code *char*) функция

Возвращает атрибут code знака *char*. Значение зависит от реализации, но в большинстве реализаций это будет ASCII-код.

(char-downcase *char*) функция

Если *char* находится в верхнем регистре, возвращает соответствующий знак из нижнего регистра; иначе возвращает *char*.

(char-greaterp *char1* &rest *chars*) функция

Действует подобно char>, но игнорирует регистр.

(char-equal *char1* &rest *chars*) функция

Действует подобно char=, но игнорирует регистр.

(char-int *char*) функция

Возвращает неотрицательное целое число, представляющее *char*. Если знак не имеет атрибутов, определенных самой реализацией, результат будет таким же, как и для char-code.

(char-lessp *char1* &rest *chars*) функция

Действует подобно char<, но игнорирует регистр.

(char-name *char*) функция

Возвращает строковое имя для *char* или nil, если оно не задано.

(char-not-greaterp *char1* &rest *chars*) функция

Действует подобно char<=, но игнорирует регистр.

(char-not-equal *char1* &rest *chars*) функция

Действует подобно char/=, но игнорирует регистр.

(char-not-lessp *char1* &rest *chars*) функция

Действует подобно char>=, но игнорирует регистр.

(char-upcase *char*) функция

Если *char* находится в нижнем регистре, преобразует его к верхнему, иначе возвращает *char*.

(char= *char1* &rest *chars*) функция

Возвращает истину, если все аргументы одинаковы.

(char/= *char1* &rest *chars*) функция

Возвращает истину, если нет ни одной пары равных аргументов.

(char< *char1* &rest *chars*) функция

Возвращает истину, если каждый последующий аргумент больше предыдущего.

(char> *char1* &rest *chars*) функция

Возвращает истину, если каждый последующий аргумент меньше предыдущего.

(char<= *char1* &rest *chars*) функция

Возвращает истину, если каждый последующий аргумент больше или равен предыдущему.

(char>= *char1* &rest *chars*) функция

Возвращает истину, если каждый последующий аргумент меньше или равен предыдущему.

(code-char *code*) функция

Возвращает знак *char*, соответствующий коду *code*.

(digit-char *i* &optional (*r* 10)) функция

Возвращает знак, представляющий *i* в системе счисления с основанием *r*.

(digit-char-p *char* &optional (*r* 10)) функция

Возвращает истину, если *char* является цифрой в системе счисления с основанием *r*.

(graphic-char-p *char*) функция

Возвращает истину, если *char* является графическим знаком.

(lower-case-p *char*) функция

Возвращает истину, если *char* находится в нижнем регистре.

(name-char *name*) функция

Возвращает знак с именем *name* (или имя *name*, если это символ). Нечувствительна к регистру.

(standard-char-p *char*) функция

Возвращает истину, если *char* является стандартным знаком.

(upper-case-p *char*) функция

Возвращает истину, если *char* находится в верхнем регистре.

Ячейки

(acons *key value alist*) функция

Эквивалент (cons (cons *key value*) *alist*).

(adjoin *object prolist* &key *key test test-not*) функция

Если member вернет истину с теми же аргументами, возвращает *prolist*; иначе вернет (cons *object prolist)*.

(append &rest *prolists*) функция

Возвращает список с элементами из всех списков *prolists*. Последний аргумент, который может быть любого типа, не копируется, поэтому (cdr (append '(a) x) будет равен (eq) x. Возвращает nil, если вызвана без аргументов.

(assoc *key alist* &key *test test-not*) функция

Возвращает первый элемент *alist*, чей car совпадает с *key*.

(assoc-if *predicate alist* &key *key*) функция

Возвращает первый элемент *alist*, для которого будет истинным *predicate*.

(assoc-if-not *predicate alist* &key *key*) [функция]

Возвращает первый элемент *alist*, для которого будет ложным *predicate*.

(atom *object*) функция

Возвращает истину, если объект *object* не является cons-ячейкой.

(butlast *list* &optional (*n* 1)) функция
(nbutlast ⟨*list*⟩ &optional (*n* 1)) функция

Возвращают копию списка *list* без последних *n* элементов или nil, если список *list* имеет менее *n* элементов. Вызывает ошибку, если *n* – отрицательное число.

(car *list*) функция

Если *list* – ячейка, возвращает ее car. Если *list* – nil, возвращает nil. Может устанавливаться с помощью setf.

(cdr *list*) функция

Если *list* – ячейка, возвращает ее cdr. Если *list* – nil, возвращает nil. Может устанавливаться с помощью setf.

(c*x*r *list*) функции

x представляет строку длиной от одного до четырех, состоящую из знаков *a* и *d*. Эквивалент соответствующего сочетания car и cdr. Например, выражение (cdaar x) эквивалентно (cdr (car (car x))). Может устанавливаться с помощью setf.

(cons *object1 object2*) функция

Возвращает новую ячейку, car которой – *object1*, а cdr – *object2*. Поэтому если *object2* является списком вида (e_1...e_n), то функция вернет (*object1* e_1...e_n).

(consp *object*) функция

Возвращает истину, если *object* является cons-ячейкой.

(copy-alist *alist*) функция

Эквивалент (mapcar #'(lambda (x) (cons (car x) (cdr x))) *alist*).

(copy-list *list*) функция

Возвращает список, равный (equal) *list*, структура верхнего уровня которого состоит из новых ячеек. Если *list* равен nil, возвращается nil.

(copy-tree *tree*) функция

Возвращает новое дерево той же формы и содержания, что и *tree*, но состоящее из новых ячеек. Если *tree* – атом, возвращается *tree*.

(endp *list*) функция

Возвращает истину, если *list* равен nil.

(first *list*)... (tenth *list*) функция

Возвращает элементы списка *list* с первого по десятый или nil, если в списке недостаточно элементов. Может устанавливаться с помощью setf.

(getf *plist key* &optional (*default* nil)) функция

Если *plist* имеет вид $(p_1\ v_1 ... p_n\ v_n)$ и p_i – первый ключ, равный (eq) *key*, возвращает v_i. Если ключ не найден, возвращает *default*. Может устанавливаться с помощью setf.

(get-properties *plist prolist*) функция

Если *plist* имеет вид $(p_1\ v_1 ... p_n\ v_n)$ и p_i – первый ключ, равный (eq) какому-либо элементу *prolist*, возвращаются p_i, v_i и $(p_i\ v_i ... p_n\ v_n)$. В противном случае возвращаются три значения nil.

(intersection *prolist1 prolist2* &key *test test-not*) функция
(nintersection ⟨*prolist1*⟩ *prolist2* &key *test test-not*) функция

Возвращают список элементов *prolist1*, принадлежащих *prolist2*. Соблюдение порядка следования элементов не гарантируется.

(last *list* &optional (*n* 1)) функция

Возвращает последние *n* ячеек в *list* или *list*, если он содержит менее *n* элементов. Если *n* равно 0, возвращает cdr последней ячейки списка *list*.

(ldiff *list object*) функция

Если *object* является хвостом *list*, возвращает новый список из элементов до *object*. В противном случае возвращает копию *list*.

(list &rest *objects*) функция

Возвращает новый список, состоящий из объектов *objects*.

(list* *object* &rest *objects*) функция

Если передан только один аргумент, возвращается он сам. В противном случае (list* $arg_1 \ldots arg_n$) эквивалентно (nconc (list $arg_1 \ldots arg_{n-1}$) arg_n).

(list-length *list*) функция

Возвращает количество ячеек в списке *list* или nil, если список является циклическим (в отличие от length, которая не определена для циклических списков). Если *list* является точечной парой, возникает ошибка.

(listp *object*) функция

Возвращает истину, когда *object* является списком, то есть cons-ячейкой или nil.

(make-list *n* &key (*initial-element* nil)) функция

Возвращает новый список из *n* элементов *initial-element*.

(mapc *function prolist* &rest *prolists*) функция

Вызывает функцию *function n* раз (*n* – длина кратчайшего *prolist*): сначала для первого элемента каждого *prolist*, затем для всех последовательных элементов каждого *prolist*, заканчивая *n*-ми. Возвращает *prolist*.

(mapcan *function prolist* &rest *prolists*) функция

Эквивалент применения nconc к результату вызова mapcar с теми же аргументами.

(mapcar *function prolist* &rest *prolists*) функция

Вызывает функцию *function n* раз (*n* – длина кратчайшего *prolist*): сначала для первого элемента каждого *prolist*, затем для всех последовательных элементов каждого *prolist*, заканчивая *n*-ми. Возвращает список значений, полученных *function*.

(mapcon *function prolist* &rest *prolists*) функция

Эквивалент применения nconc к результату вызова maplist с теми же аргументами.

(mapl *function prolist* &rest *prolists*) функция

Вызывает функцию *function n* раз (*n* – длина кратчайшего *prolist*): сначала для каждого *prolist* целиком, затем для всех последовательных cdr каждого *prolist*, заканчивая (*n*–1)-м. Возвращает *prolist*.

(maplist *function prolist* &rest *prolists*) функция

Вызывает функцию *function n* раз (*n* – длина кратчайшего *prolist*): сначала для каждого *prolist* целиком, затем для всех последовательных cdr каждого *prolist*, заканчивая (*n*–1)-м. Возвращает список значений, полученных *function*.

(member *object prolist* &key *key test test-not*) функция

Возвращает хвост *prolist*, начинающийся с первого элемента, совпадающего с *object*, или nil, если совпадений не найдено.

(member-if *predicate prolist* &key *key test test-not*) функция

Возвращает хвост *prolist*, начинающийся с первого элемента, для которого *predicate* вернул истину, или nil, если совпадений не найдено.

(member-if-not *predicate prolist* &key *key test test-not*) [функция]

Возвращает хвост *prolist*, начинающийся с первого элемента, для которого *predicate* вернул ложь, или nil, если совпадений не найдено.

(nconc &rest ⟨*lists*⟩) функция

Возвращает список с элементами из всех *lists* (по порядку). Принцип работы: устанавливает cdr последней ячейки каждого списка в начало следующего списка. Последний аргумент может быть объектом любого типа. Вызванная без аргументов, возвращает nil.

(nth *n list*) функция

Возвращает (n+1)-й элемент списка *list*. Возвращает nil, если список *list* имеет менее (n+1) элементов. Может устанавливаться с помощью setf.

(nthcdr *n list*) функция

Эквивалент n-кратного последовательного применения cdr к *list*.

(null *object*) функция

Возвращает истину, если объект *object* является nil.

(pairlis *keys values* &optional *alist*) функция

Возвращает то же значение, что и (nconc (mapcar #'cons *keys values*) *alist*) или (nconc (nreverse (mapcar #'cons *keys values*)) *alist*). Требует, чтобы *keys* и *values* были одинаковой длины.

(pop ⟨*place*⟩) макрос

Устанавливает значение места *place*, которое должно вычисляться в списке *list*, в (cdr *list*). Возвращает (car *list*).

(push *object* ⟨*place*⟩) макрос

Устанавливает значение места *place* в (cons *object place*). Возвращает это значение.

(pushnew *object* ⟨*place*⟩ &key *key test test-not*) макрос

Устанавливает значение места *place*, которое должно вычисляться в правильный список, в результат вызова adjoin с теми же аргументами. Возвращает новое значение *place*.

(rassoc *key alist* &key *key test test-not*) функция

Возвращает первый элемент *alist*, cdr которого совпал с *key*.

(rassoc-if *predicate alist* &key *key*) функция

Возвращает первый элемент *alist*, для cdr которого *predicate* вернул истину.

(rassoc-if-not *predicate alist* &key *key*) функция

Возвращает первый элемент *alist*, для cdr которого *predicate* вернул ложь.

(remf ⟨*place*⟩ *key*) макрос

Первый аргумент, *place*, должен вычисляться в список свойств *plist*. Если *plist* имеет вид (p_1 v_1...p_n v_n) и p_i – первый *p*, равный (eq) *key*, деструктивно удаляет p_i и v_i из *plist* и записывает результат в *place*. Возвращает истину, если что-либо было удалено, в противном случае возвращает ложь.

(rest *list*) функция

Идентична cdr. Может устанавливаться с помощью setf.

(revappend *list1 list2*) функция
(nreconc ⟨*list1*⟩ *list2*) функция

Эквивалентны (nconc (reverse *list1*) *list2*) и (nconc (nreverse *list1*) *list2*) соответственно.

(rplaca ⟨*cons*⟩ *object*) функция

Эквивалент (setf (car *cons*) *object*), но возвращает *cons*.

(rplacd ⟨*cons*⟩ *object*) функция

Эквивалент (setf (cdr *cons*) *object*), но возвращает *cons*.

(set-difference *prolist1 prolist2* &key *key test test-not*) функция
(nset-difference ⟨*prolist1*⟩ *prolist2* &key *key test test-not*) функция

Возвращают список элементов *prolist1*, не имеющихся в *prolist2*. Сохранение порядка следования элементов не гарантируется.

(set-exclusive-or *prolist1 prolist2* &key *key test test-not*) функция
(nset-exclusive-or ⟨*prolist1*⟩ *prolist2* &key *key test test-not*) функция

Возвращают список элементов, имеющихся либо в *prolist1*, либо в *prolist2*, но не в обоих списках одновременно. Сохранение порядка следования элементов не гарантируется.

(sublis *alist tree* &key *key test test-not*) функция
(nsublis *alist* ⟨*tree*⟩ &key *key test test-not*) функция

Возвращают дерево, подобное *tree*, но каждое поддерево, совпадающее с ключом в *alist*, заменяется на соответствующее ему значение. Если не было произведено никаких модификаций, возвращается *tree*.

(subsetp *prolist1 prolist2* &key *key test test-not*) функция

Возвращает истину, если каждый элемент *prolist1* имеется в *prolist2*.

(subst *new old tree* &key *key test test-not*) функция
(nsubst *new old* ⟨*tree*⟩ &key *key test test-not*) функция

Возвращают дерево, подобное *tree*, в котором все поддеревья, совпадающие с *old*, заменены на *new*.

(subst-if *new predicate tree* &key *key*) функция

(nsubst-if *new predicate* ⟨*tree*⟩ &key *key*) функция

Возвращают дерево, подобное *tree*, в котором все поддеревья, для которых справедлив предикат *predicate*, заменены на *new*.

(subst-if-not *new predicate tree* &key *key*) [функция]
(nsubst-if-not *new predicate* ⟨*tree*⟩ &key *key*) [функция]

Возвращают дерево, подобное *tree*, в котором все поддеревья, для которых не справедлив предикат *predicate*, заменены на *new*.

(tailp *object list*) функция

Возвращает истину, когда *object* является хвостом *list*, то есть когда *object* либо nil, либо одна из ячеек, из которых состоит *list*.

(tree-equal *tree1 tree2* &key *test test-not*) функция

Возвращает истину, когда *tree1* и *tree2* имеют одинаковую форму и все их элементы совпадают.

(union *prolist1 prolist2* &key *key test test-not*) функция
(nunion ⟨*prolist1*⟩ ⟨*prolist2*⟩ &key *key test test-not*) функция

Возвращают список элементов, имеющихся в *prolist1* и *prolist2*. Сохранение порядка следования элементов в возвращаемом списке не гарантируется. Если либо *prolist1*, либо *prolist2* имеет повторяющиеся элементы, то эти элементы будут дублироваться в возвращаемом списке.

Массивы

(adjustable-array-p *array*) функция

Возвращает истину, если массив *array* расширяемый.

(adjust-array ⟨*array*⟩ *dimensions* &key ...) функция

Возвращает массив, подобный *array* (или сам *array*, если он расширяемый) с некоторыми измененными свойствами. Если одна из размерностей *dimensions* уменьшается, исходный массив обрезается; если увеличивается, то новые элементы заполняются согласно параметру :initial-element. Ключи те же, что и для make-array, со следующими дополнительными ограничениями:

:element-type *type*

Тип *type* должен быть совместимым с исходным.

`:initial-element` *object*

Новые элементы будут равны *object*, старые сохранят исходные значения.

`:initial-contents` *seq*

Как и в `make-array`, существующие элементы *array* будут перезаписаны.

`:fill-pointer` *object*

Если *object* является `nil`, указатель заполнения (если имеется) останется прежним.

`:displaced-to` *array2*

Если исходный массив *array* является предразмещенным, а *array2* – `nil`, то соответствующие элементы оригинального массива будут скопированы в возвращаемый массив, заполняя все неустановленные значения (если таковые имеются) элементами `:initial-element` (если задан). Если исходный массив не является предразмещенным и *array2* – массив, тогда исходное значение будет потеряно и новый массив будет размещен поверх *array2*. В противном случае действует так же, как `make-array`.

`:displaced-index-offset` *i*

Если этот аргумент не передается, предразмещенный массив не будет смещен относительно оригинала.

(`aref` *array* `&rest` *is*) функция

Возвращает элемент массива *array* с индексами *is* (или один элемент этого массива, если размерность массива равна нулю и индексы *is* не заданы). Игнорирует указатели заполнения. Может устанавливаться с помощью `setf`.

(`array-dimension` *array i*) функция

Возвращает *i*-ю размерность массива. Индексирование начинается с нуля.

(`array-dimensions` *array*) функция

Возвращает список целых чисел, представляющих все размерности массива.

(`array-displacement` *array*) функция

Возвращает два значения: массив, поверх которого предразмещен *array*, и его смещение. Возвращает `nil` и `0`, если массив *array* не предразмещенный.

(`array-element-type` *array*) функция

Возвращает тип элементов массива *array*.

(`array-has-fill-pointer-p` *array*) функция

Возвращает истину, если массив *array* имеет указатель заполнения.

(array-in-bounds-p *array* &rest *is*) функция

Возвращает истину, если те же аргументы будут корректными аргументами aref.

(arrayp *object*) функция

Возвращает истину, если *object* – массив.

(array-rank *array*) функция

Возвращает количество размерностей массива *array*.

(array-row-major-index *array* &rest *is*) функция

Возвращает номер элемента для заданных индексов *is*, считая массив *array* одномерным и расположенным построчно. Индексирование начинается с нуля.

(array-total-size *array*) функция

Возвращает количество элементов в массиве *array*.

(bit *bit-array* &rest *is*) функция

Аналог aref для бит-массива *bit-array*. Может устанавливаться с помощью setf.

(bit-and ⟨*bit-array1*⟩ *bit-array2* &optional ⟨*arg*⟩) функция

Работает с бит-массивами по аналогии с logand для целых чисел: выполняет логическое И для двух бит-массивов одинаковой размерности, возвращая итоговый массив. Если *arg* равен t, возвращается новый массив; если *arg* – nil, изменяется существующий *bit-array1*; если *arg* – бит-массив (такой же размерности), используется он сам.

(bit-andc1 ⟨*bit-array1*⟩ *bit-array2* &optional ⟨*arg*⟩) функция

Аналог bit-and, только основан на logandc1.

(bit-andc2 ⟨*bit-array1*⟩ *bit-array2* &optional ⟨*arg*⟩) функция

Аналог bit-and, только основан на logandc2.

(bit-eqv ⟨*bit-array1*⟩ *bit-array2* &optional ⟨*arg*⟩) функция

Аналог bit-and, только основан на logaeqv.

(bit-ior ⟨*bit-array1*⟩ *bit-array2* &optional ⟨*arg*⟩) функция

Аналог bit-and, только основан на logior.

(bit-nand ⟨*bit-array1*⟩ *bit-array2* &optional ⟨*arg*⟩) функция

Аналог bit-and, только основан на lognand.

(bit-nor ⟨*bit-array1*⟩ *bit-array2* &optional ⟨*arg*⟩) функция

Аналог bit-and, только основан на lognor.

(bit-not ⟨*bit-array*⟩ &optional ⟨*arg*⟩) функция

Работает с бит-массивами по аналогии с lognot для целых чисел: возвращает логическое дополнение до *bit-array*. Если *arg* – t, возвраща-

ется новый массив; если *arg* – `nil`, изменяется существующий *bit-array1*; если *arg* – бит-массив (такой же размерности), используется он сам.

(`bit-orc1` ⟨*bit-array1*⟩ *bit-array2* `&optional` ⟨*arg*⟩) функция

Аналог `bit-and`, только основан на `logorc1`.

(`bit-orc2` ⟨*bit-array1*⟩ *bit-array2* `&optional` ⟨*arg*⟩) функция

Аналог `bit-and`, только основан на `logorc2`.

(`bit-xor` ⟨*bit-array1*⟩ *bit-array2* `&optional` ⟨*arg*⟩) функция

Аналог `bit-and`, только основан на `logxor`.

(`bit-vector-p` *object*) функция

Возвращает истину, если объект *object* является бит-вектором.

(`fill-pointer` *vector*) функция

Возвращает указатель заполнения вектора *vector*. Может устанавливаться с помощью `setf`, но лишь в тех случаях, когда вектор уже имеет указатель заполнения.

(`make-array` *dimensions* `&key` *element-type initial-element initial-contents adjustable fill-pointer displaced-to displaced-index-offset*) функция

Возвращает новый массив с размерностями *dimensions* (или создается вектор заданной длины, если *dimensions* – одно число). По умолчанию элементы могут иметь любой тип, если не задан конкретный тип. Доступны следующие аргументы по ключу:

`:element-type` *type*

Декларирует тип *type* элементов массива.

`:initial-element` *object*

Каждым элементом создаваемого массива будет *object*. Не может использоваться вместе с `:initial-contents`.

`:initial-contents` *seq*

Элементами массива будут соответствующие элементы набора вложенных последовательностей *seq*. Для массива с нулевой размерностью может использоваться одиночный аргумент. Не может использоваться вместе с `:initial-element`.

`:adjustable` *object*

Если *object* истинен, создаваемый массив гарантированно будет расширяемым; впрочем, он может быть таковым и в противном случае.

`:fill-pointer` *object*

Если *object* истинен, массив (должен быть вектором) будет иметь указатель заполнения. Если *object* является целым числом из

промежутка от нуля до длины вектора, он будет исходным значением указателя.

`:displaced-to` *array*

Массив будет предразмещен поверх *array*. Ссылки на его элементы будут транслироваться в ссылки на соответствующие элементы *array* (элементы в тех же позициях, что и у переданного массива, элементы которого расположены построчно).

`:displaced-index-offset` *i*

Смещение предразмещения поверх другого массива будет равно *i*. Может задаваться лишь совместно с `:displaced-to`.

(`row-major-aref` *array i*) функция

Возвращает *i*-й элемент массива *array*, рассматриваемого как одномерный с построчным заполнением. Индексирование начинается с нуля. Может устанавливаться с помощью `setf`.

(`sbit` *simple-bit-array* `&rest` *is*) функция

Аналог `aref` для простых бит-массивов. Может устанавливаться с помощью `setf`.

(`simple-bit-vector-p` *object*) функция

Возвращает истину, если *object* является простым бит-массивом.

(`simple-vector-p` *object*) функция

Возвращает истину, если *object* является простым вектором.

(`svref` *simple-vector i*) функция

Возвращает *i*-й элемент *simple-vector*. Индексирование начинается с нуля. Значение может устанавливаться с помощью `setf`.

(`upgraded-array-element-type` *type* `&optional` *env*) функция

Возвращает реальный тип элементов, который реализация может передать массиву, чей `:element-type` был объявлен как *type*.

(`vector &rest` *objects*) функция

Возвращает новый простой вектор с элементами *objects*.

(`vectorp &rest` *objects*) функция

Возвращает истину, если *object* является вектором.

(`vector-pop` *vector*) функция

Уменьшает индекс заполнения вектора *vector* и возвращает элемент, на который он указывал. Попытка применения к вектору без указателя заполнения или с указателем, равным `0`, приводит к ошибке.

(`vector-push` *object vector*) функция

Если указатель заполнения равен длине вектора *vector*, возвращает `nil`. В противном случае замещает элемент, на который указывает

указатель заполнения, на *object*, затем увеличивает указатель заполнения и возвращает старое значение. Попытка применения к вектору без указателя заполнения приводит к ошибке.

(vector-push-extend *object vector* &optional *i*) функция

Аналог vector-push, но если указатель заполнения равен длине вектора *vector*, вектор предварительно удлиняется на *i* элементов (или на значение по умолчанию, зависящее от реализации) с помощью вызова adjust-array.

Строки

(char *string i*) функция

Возвращает *i*-й знак строки *string*. Индексация начинается с нуля. Игнорирует указатели заполнения. Может устанавливаться с помощью setf.

(make-string *n* &key *initial-element* (*element-type* 'character)) функция

Возвращает новую строку из *n* элементов *initial-element* (значение по умолчанию зависит от реализации).

(schar *simple-string i*) функция

Действует подобно char, но применима только к простым строкам. Может устанавливаться с помощью setf.

(simple-string-p *object*) функция

Возвращает истину, если *object* является простой строкой.

(string *arg*) функция

Если *arg* – строка, она возвращается; если это символ, то возвращается его имя; если знак, возвращается строка из этого знака.

(string-capitalize *string* &key *start end*) функция
(nstring-capitalize ⟨*string*⟩ &key *start end*) функция

Возвращают строку, в которой первая буква каждого слова находится в верхнем регистре, а все остальные – в нижнем. Словом считается любая последовательность алфавитных символов. Первым аргументом string-capitalize может быть символ, и в этом случае будет использоваться его имя.

(string-downcase *string* &key *start end*) функция
(nstring-downcase ⟨*string*⟩ &key *start end*) функция

Действуют подобно string-upcase и nstring-upcase, но все знаки строки *string* преобразуются к нижнему регистру.

(string-equal *string1 string2* &key *start1 end1 start2 end2*) функция

Действует подобно string=, но игнорирует регистр.

(string-greaterp *string1 string2* &key *start1 end1 start2 end2*) функция

Действует подобно string>, но игнорирует регистр.

(string-upcase *string* &key *start end*) функция
(nstring-upcase ⟨*string*⟩ &key *start end*) функция

Возвращают строку, в которой все знаки в нижнем регистре преобразованы к соответствующим знакам в верхнем регистре. Параметры *start* и *end* действуют так же, как и в других строковых функциях. Первый аргумент string-upcase может быть символом, и в этом случае используется его имя.

(string-left-trim *seq string*) функция

Действует подобно string-trim, но обрезает строку только с начала.

(string-lessp *string1 string2* &key *start1 end1 start2 end2*) функция

Действует подобно string<, но игнорирует регистр.

(string-not-equal *string1 string2* &key *start1 end1 start2 end2*) функция

Действует подобно string/=, но игнорирует регистр.

(string-not-greaterp *string1 string2* &key *start1 end1 start2 end2*) функция

Действует подобно string<=, но игнорирует регистр.

(string-not-lessp *string1 string2* &key *start1 end1 start2 end2*) функция

Действует подобно string>=, но игнорирует регистр.

(stringp *object*) функция

Возвращает истину, если *object* является строкой.

(string-right-trim *seq string*) функция

Действует подобно string-trim, но обрезает строку только с конца.

(string-trim *seq string*) функция

Возвращает строку, подобную *string*, в которой все знаки, перечисленные в *seq*, удалены с обоих концов.

(string= *string1 string2* &key *start1 end1 start2 end2*) функция

Возвращает истину, если подпоследовательности *string1* и *string2* имеют одну длину и содержат одинаковые знаки. Параметры *start1* и *end1*, *start2* и *end2* работают как обычные *start* и *end* для *string1* и *string2* соответственно.

(string/= *string1 string2* &key *start1 end1 start2 end2*) функция

Возвращает истину, если string= возвращает ложь.

(string< *string1 string2* &key *start1 end1 start2 end2*) функция

Возвращает истину, если две подпоследовательности содержат одинаковые знаки до конца первой последовательности и вторая длиннее первой; или если подпоследовательности имеют разные знаки и для первой отличающейся пары знаков справедлив char<. Используются те же параметры, что и в string=.

(string> *string1 string2* &key *start1 end1 start2 end2*) функция

Возвращает истину, если две подпоследовательности содержат одинаковые знаки до конца второй последовательности и первая длиннее второй; или если подпоследовательности имеют разные знаки и для первой отличающейся пары знаков справедлив char>. Используются те же параметры, что и в string=.

(string<= *string1 string2* &key *start1 end1 start2 end2*) функция

Возвращает истину, если истинны string< либо string= с теми же аргументами.

(string>= *string1 string2* &key *start1 end1 start2 end2*) функция

Возвращает истину, если истинны string> либо string= с теми же аргументами.

Последовательности

(concatenate *type* &rest *sequences*) функция

Возвращает новую последовательность типа *type* со следующими по порядку элементами последовательностей *sequences*. Копирует каждую последовательность, даже последнюю.

(copy-seq *proseq*) функция

Возвращает новую последовательность того же типа и с теми же элементами, что и *proseq*.

(count *object proseq* &key *key test test-not from-end start end*) функция

Возвращает количество совпадающих с *object* элементов *proseq*.

(count-if *predicate proseq* &key *key from-end start end*) функция

Возвращает количество элементов *proseq*, соответствующих предикату *predicate*.

(count-if-not *predicate proseq* &key *key from-end start end*) [функция]

Возвращает количество не соответствующих предикату *predicate* элементов *proseq*.

(elt *proseq n*) функция

Возвращает (n+1)-й элемент *proseq*. Если длина *proseq* меньше n+1, вызывается ошибка. Может устанавливаться с помощью setf.

(fill ⟨*proseq*⟩ *object* &key *start end*) функция

Деструктивно заполняет *proseq* элементами *object*. Возвращает *proseq*.

(find *object proseq* &key *key test test-not from-end start end*) функция

Возвращает первый совпадающий с *object* элемент *proseq*.

(find-if *predicate proseq* &key *key from-end start end*) функция

Возвращает первый соответствующий предикату элемент *proseq*.

(find-if-not *predicate proseq* &key *key from-end start end*) [функция]

Возвращает первый не соответствующий предикату элемент *proseq*.

(length *proseq*) функция

Возвращает количество элементов в последовательности *proseq*. Если *proseq* имеет указатель заполнения, учитываются лишь элементы, идущие до него.

(make-sequence *type n* &key (*initial-element* nil)) функция

Возвращает новую последовательность типа *type* из *n* элементов *initial-element*.

(map *type function proseq* &rest *proseqs*) функция

Вызывает функцию *function n* раз (*n* – длина кратчайшей *proseq*): один раз для первого элемента каждой *proseq*, затем для всех элементов вплоть до *n*-х элементов каждой *proseq*. Возвращает список значений типа *type*, полученных *function*. (Если *type* является nil, то функция ведет себя как mapc для последовательностей.)

(map-into ⟨*result*⟩ *function proseq* &rest *proseqs*) функция

Вызывает функцию *function n* раз (*n* – длина кратчайшей *proseq*, включая *result*): один раз для первого элемента каждой *proseq*, затем для всех элементов вплоть до *n*-х элементов каждой *proseq*, Деструктивно заменяет первые *n* элементов *result* на значения, возвращаемые *function*, и возвращает *result*.

(merge *type* ⟨*sequence1*⟩ ⟨*sequence2*⟩ *predicate* &key *key*) функция

Эквивалентна (stable-sort (concatenate *type sequence1 sequence2*) *predicate* :key *key*), но деструктивна и более эффективна.

(mismatch *sequence1 sequence2* функция
&key *key test test-not from-end*
start1 end1 start2 end2)

Возвращает положение (индексация начинается с нуля) первого элемента *sequence1*, с которого начинается различие между *sequence1* и *sequence2*. Если последовательности полностью совпадают, возвращается nil. Параметры *start1* и *end1*, *start2* и *end2* работают как обычные *start* и *end* для *sequence1* и *sequence2* соответственно.

(position *object proseq* функция
&key *key test test-not from-end start end*)

Возвращает положение (индексация начинается с нуля) первого элемента *proseq*, совпадающего с *object*.

(position-if *predicate proseq* &key *key from-end start end*) функция

Возвращает положение (индексация начинается с нуля) первого элемента *proseq*, для которого будет справедлив предикат *predicate*.

(position-if-not *predicate proseq* [функция]
&key *key from-end start end*)

Возвращает положение (индексация начинается с нуля) первого элемента *proseq*, для которого будет несправедлив предикат *predicate*.

(reduce *function proseq* функция
&key *key from-end start end initial-value*)

Поведение reduce описывается в следующей таблице, где *f* – функция, а элементами *proseq* являются *a*, *b* и *c*:

from-end	*initial-value*	**эквивалент**
ложь	нет	(*f* (*f a b*) *c*)
ложь	да	(*f* (*f* (*f initial-value a*) *b*) *c*)
истина	нет	(*f a* (*f b c*))
истина	да	(*f a* (*f b* (*f c initial-value*)))

Если *proseq* состоит из одного элемента и значение *initial-value* не предоставлено, возвращается сам элемент. Если *proseq* пуста и предоставлено *initial-value*, оно возвращается, но если *initial-value* не предоставлено, функция вызывается без аргументов. Если предоставлены *key* и *initial-value*, *key* не применяется к *initial-value*.

(remove *object proseq* функция
&key *key test test-not from-end start end count*)

(delete *object* ⟨*proseq*⟩ функция
&key *key test test-not from-end start end count*)

Возвращают последовательность, похожую на *proseq*, но без всех элементов, совпадающих с *object*. Если предоставлен *count*, удаляются лишь первые *count* экземпляров.

(remove-duplicates *proseq* функция
&key *key test test-not from-end start end*)

(delete-duplicates ⟨*proseq*⟩ функция
&key *key test test-not from-end start end*)

Возвращают последовательность, похожую на *proseq*, из которой удалены все повторяющиеся элементы, кроме последнего.

(remove-if *predicate proseq* функция
&key *key from-end start end count*)

(delete-if *predicate* ⟨*proseq*⟩ функция
&key *key from-end start end count*)

Возвращают последовательность, похожую на *proseq*, но без всех элементов, для которых справедлив предикат *predicate*. Если предоставлен *count*, удаляются лишь первые *count* экземпляров.

(remove-if-not *predicate proseq* [функция]
&key *key from-end start end count*)

(delete-if-not *predicate* ⟨*proseq*⟩ [функция]
&key *key from-end start end count*)

Возвращают последовательность, похожую на *proseq*, но без всех элементов, для которых несправедлив предикат *predicate*. Если предоставлен *count*, удаляются лишь первые *count* экземпляров.

(replace ⟨*sequence1*⟩ *sequence2* функция
&key *start1 end1 start2 end2*)

Деструктивно заменяет *sequence1* на *sequence2* и возвращает *sequence1*. Количество заменяемых элементов равно длине кратчайшей последовательности. Работает, только если последовательности равны (eq), и не работает, когда они всего лишь разделяют общую структуру (в этом случае результат перезаписи не определен).

(reverse *proseq*) функция
(nreverse ⟨*proseq*⟩) функция

Возвращают последовательность того же типа, что и *proseq*, содержащую те же элементы в обратном порядке. Последовательность, возвращаемая reverse, всегда является копией. Если *proseq* – вектор, reverse возвращает простой вектор.

(search *sequence1 sequence2* функция
&key *key test test-not from-end start1 end1 start2 end2*)

Возвращает положение (индексация начинается с нуля) начала первой совпадающей с *sequence1* подпоследовательности в *sequence2*. Если не найдено ни одной совпадающей подпоследовательности, возвращается nil.

(sort ⟨*proseq*⟩ *predicate* &key *key*) функция

Возвращает последовательность того же типа, что и *proseq*, и с теми же элементами, отсортированными так, что для любых двух последующих элементов *e* и *f* значение (*predicate e f*) ложно, а (*predicate f e*) – истинно.

(stable-sort ⟨*proseq*⟩ *predicate* &key *key*) функция

Действует подобно sort, но старается максимально сохранить порядок следования элементов *proseq*.

(subseq *proseq start* &optional *end*) функция

Возвращает новую последовательность, являющуюся подпоследовательностью *proseq*. Параметры *start* и *end* ограничивают положение подпоследовательности: *start* указывает на положение (индексация начинается с нуля) первого элемента подпоследовательности, а *end*, если задан, указывает положение *после* последнего элемента подпоследовательности. Может устанавливаться с помощью setf, а также replace.

(substitute *new old proseq* &key *key test test-not from-end start end count*) функция

(nsubstitute *new old* ⟨*proseq*⟩ &key *key test test-not from-end start end count*) функция

Возвращают последовательность, похожую на *proseq*, в которой все элементы, совпадающие с *old*, заменяются на *new*. Если предоставляется *count*, заменяются только первые *count* совпавших элементов.

(substitute-if *new predicate proseq* &key *key from-end start end count*) функция

(nsubstitute-if *new predicate* ⟨*proseq*⟩ &key *key from-end start end count*) функция

Возвращают последовательность, похожую на *proseq*, в которой все элементы, для которых справедлив *predicate*, заменяются на *new*. Если предоставляется *count*, заменяются только первые *count* совпавших элементов.

(substitute-if-not *new predicate proseq* &key *key from-end start end count*) [функция]

(nsubstitute-if-not *new predicate* ⟨*proseq*⟩ &key *key from-end start end count*) [функция]

Возвращают последовательность, похожую на *proseq*, в которой все элементы, для которых несправедлив *predicate*, заменяются на *new*. Если предоставляется *count*, заменяются только первые *count* совпавших элементов.

Хеш-таблицы

(clrhash *hash-table*) функция

Удаляет все элементы, хранящиеся в таблице *hash-table*, и возвращает ее.

(gethash *key hash-table* &optional *default*) функция

Возвращает объект, соответствующий ключу *key* в таблице *hash-table*, или *default*, если заданный ключ не найден. Второе возвращаемое значение истинно, если искомый элемент был найден. Может устанавливаться с помощью setf.

(hash-table-count *hash-table*) функция

Возвращает количество элементов в *hash-table*.

(hash-table-p *object*) функция

Возвращает истину, если *object* является хеш-таблицей.

(hash-table-rehash-size *hash-table*) функция

Возвращает число, имеющее такое же значение, что и параметр :rehash-size в make-hash-table. Оно показывает, насколько вырастет размер *hash-table* при ее расширении.

(hash-table-rehash-threshold *hash-table*) функция

Возвращает число, имеющее такое же значение, что и параметр :rehash-threshold в make-hash-table. Оно указывает число элементов, при достижении которого *hash-table* будет расширена.

(hash-table-size *hash-table*) функция

Возвращает емкость *hash-table*.

(hash-table-test *hash-table*) функция

Возвращает функцию, используемую для определения равенства ключей в *hash-table*.

(make-hash-table &key *test size rehash-size rehash-threshold*) функция

Возвращает новую хеш-таблицу, использующую *test* (по умолчанию eql) для установления равенства ключей. Размер *size* является предполагаемой емкостью создаваемой таблицы. *rehash-size* является мерой объема, добавляемого к таблице при ее расширении (если integer, то объем складывается, если float, то умножается). Положительное число *rehash-threshold* определяет степень заполнения, начиная с которой будет разрешено расширение таблицы.

(maphash *function hash-table*) функция

Применяет *function*, которая должна быть функцией двух аргументов, к каждому ключу и соответствующему ему значению в *hash-table*.

(remhash *key* ⟨*hash-table*⟩) функция

Удаляет из *hash-table* объект, связанный с ключом *key*. Возвращает истину, если ключ был найден.

(sxhash *object*) функция

Является, по сути, хеширующей функцией для equal-хеш-таблиц. Возвращает уникальное неотрицательное число типа fixnum для каждого набора равных (equal) аргументов.

(with-hash-table-iterator (*symbol hash-table*) *declaration* expression**) макрос

Вычисляет выражения *expressions*, связывая *symbol* с локальным макросом, последовательно возвращающим информацию об элементах *hash-table*. Локальный макрос возвращает три значения: значение, указывающее, возвращаются ли другие значения (так nil говорит о том, что мы дошли до конца таблицы), ключ и соответствующее ему значение.

Пути к файлам

(directory-namestring *path*) функция

Возвращает зависящую от реализации строку, представляющую компоненту directory пути *path*.

(enough-namestring *path* &optional *path2*) функция

Возвращает зависящую от реализации строку, достаточную для идентификации пути *path* относительно *path2* (как если бы *path2* был *default-pathname-defaults*).

(file-namestring *path*) функция

Возвращает зависящую от реализации строку, представляющую компоненты name, type и version пути *path*.

(host-namestring *path*) функция

Возвращает зависящую от реализации строку, представляющую компоненту host пути *path*.

(load-logical-pathname-translations *string*) функция

Загружает определение логического хоста с именем *string*, если оно до сих пор не загружено. Возвращает истину, если что-либо было загружено.

(logical-pathname *path*) функция

Возвращает логический путь, соответствующий *path*.

(logical-pathname-translations *host*) функция

Возвращает список преобразований для *host*, который должен быть логическим хостом или указывающей на него строкой.

(make-pathname &key *host device directory name type version defaults case*) функция

Возвращает путь, составленный из своих аргументов. Значения незаданных аргументов берутся из *defaults*; если он не предоставляется, хостом по умолчанию считается хост из *default-pathname-defaults*, остальные компоненты по умолчанию имеют значение nil.

Хост *host* может быть строкой или списком строк. Устройство *device* может быть строкой. Каталог *directory* может быть строкой, списком строк или `:wild`. Имя *name* и тип *type* могут быть строками или `:wild`. Версия *version* может быть неотрицательным целым числом, `:wild` или `:newest` (в большинстве реализаций также доступны `:oldest`, `:previous` или `:installed`). Все вышеперечисленные параметры могут также быть `nil` (в этом случае компонента получает значение по умолчанию) или `:unspecific`, что означает «неприменимость» данной компоненты (поддержка этого варианта зависит от конкретной операционной системы).

Аргумент *defaults* может быть любым допустимым (valid) аргументом *pathname* и расценивается подобным образом. Компоненты, которые не заданы или заданы как `nil`, получают свои значения из параметра *defaults*, если он задан.

Если *case* задан как `:local` (по умолчанию), компоненты пути будут в регистре локальной системы; если *case* задан как `:common`, то компоненты, находящиеся полностью в верхнем регистре, используются так, как принято в данной системе, а компоненты в смешанном регистре используются без изменений.

(`merge-pathnames` *path* `&optional` *default-path version*) функция

Возвращает путь, полученный заполнением всех недостающих компонент в *path* с помощью *default-path* (по умолчанию `*default-pathname-defaults*`). Если *path* включает компоненту `name`, версия может быть взята из *version* (по умолчанию `:newest`); в противном случае она берется из *default-path*.

(`namestring` *path*) функция

Возвращает зависящую от реализации строку, представляющую путь *path*.

(`parse-namestring` *path* `&optional` *host default* `&key` *start end junk-allowed*) функция

Если путь *path* не является строкой, возвращает соответствующий путь в обычном виде. Если задана строка, обрабатывает ее как логический путь, получая хост из параметра *host* или из самой строки, или из пути *default* (в приведенном порядке). Если не найден допустимый (valid) путь и не задан *junk-allowed*, генерирует ошибку. Если *junk-allowed* задан, то возвращает `nil`. Параметры *start* и *end* используются по аналогии со строковыми функциями. Вторым значением возвращает положение в строке, на котором завершилось ее чтение.

(`pathname` *path*) функция

Возвращает путь, соответствующий *path*.

(`pathname-host` *path* `&key` *case*) функция

Возвращает компоненту `host` пути *path*. Параметр `:case` обрабатывается так же, как в `make-pathname`.

(pathname-device *path* &key *case*) функция

Возвращает компоненту device пути *path*. Параметр :case обрабатывается так же, как в make-pathname.

(pathname-directory *path* &key *case*) функция

Возвращает компоненту directory пути *path*. Параметр :case обрабатывается так же, как в make-pathname.

(pathname-match-p *path wild-path*) функция

Возвращает истину, если *path* совпадает с *wild-path*; любой отсутствующий компонент *wild-path* рассматривается как :wild.

(pathname-name *path* &key *case*) функция

Возвращает компоненту name пути *path*. Параметр :case обрабатывается так же, как в make-pathname.

(pathnamep *object*) функция

Возвращает истину, если *object* является путем.

(pathname-type *path* &key *case*) функция

Возвращает компоненту type пути *path*. Параметр :case обрабатывается так же, как в make-pathname.

(pathname-version *path* &key *case*) функция

Возвращает компоненту version пути *path*. Параметр :case обрабатывается так же, как в make-pathname.

(translate-logical-pathname *path* &key) функция

Возвращает физический путь, соответствующий *path*.

(translate-pathname *path1 path2 path3* &key) функция

Преобразовывает путь *path1*, совпадающий с wild-путем *path2*, в соответствующий путь, совпадающий с wild-путем *path3*.

(wild-pathname-p *path* &optional *component*) функция

Возвращает истину, если компонента пути, на которую указывает *component* (может быть :host, :device, :directory, :name, :type или :version), является wild или (если *component* задан как nil) *path* содержит какие-либо wild-компоненты.

Файлы

(delete-file *path*) функция

Удаляет файл, на который указывает *path*. Возвращает t.

(directory *path* &key) функция

Создает и возвращает список путей, соответствующих реальным файлам, совпадающим с шаблоном *path* (который может содержать wild-компоненты).

(ensure-directories-exist *path* &key *verbose*) функция

Если каталоги, в которых находится файл *path*, не существуют, пытается создать их (информируя об этом, если задан *verbose*). Возвращает два значения: *path* и еще одно, истинное, если были созданы какие-либо каталоги.

(file-author *path*) функция

Возвращает строку, представляющую автора (владельца) файла *path*, или nil, если владелец не может быть установлен.

(file-error-pathname *condition*) функция

Возвращает путь, который привел к исключению *condition*.

(file-write-date *path*) функция

Возвращает время (в том же формате, что и get-universal-time), соответствующее времени последнего изменения файла *path*, или nil, если время не может быть установлено.

(probe-file *path*) функция

Возвращает актуальное имя файла, на который указывает *path*, или nil, если файл не найден.

(rename-file *path1* *path2*) функция

Переименовывает файл *path1* в *path2* (который не может быть потоком). Незаданные компоненты в *file2* по умолчанию заполняются из *path1*. Возвращает три значения: полученный путь, актуальное имя старого файла и актуальное имя нового файла.

(truename *path*) функция

Возвращает актуальное имя файла, на который указывает *path*. Если файл не найден, вызывает ошибку.

Потоки

(broadcast-stream-streams *broadcast-stream*) функция

Возвращает список потоков, из которых создан *broadcast-stream*.

(clear-input &optional *stream*) функция

Очищает любой ожидающий во входном потоке *stream* ввод и возвращает nil.

(clear-output &optional *stream*) функция

Очищает любой ожидающий в буфере вывод для *stream* и возвращает nil.

(close *stream* &key *abort*) функция

Закрывает поток *stream*, возвращая t, если поток был открытым. Если задан *abort*, пытается устранить все побочные эффекты от соз-

дания потока (например, связанный с ним выходной файл будет удален). Запись в закрытый поток невозможна, но он может передаваться вместо пути в другие функции, включая `open`.

(`concatenated-stream-streams` *concatenated-stream*) функция

Возвращает упорядоченный список потоков, из которых *concatenated-stream* будет осуществлять чтение.

(`echo-stream-input-stream` *echo-stream*) функция

Возвращает поток ввода для *echo-stream*.

(`echo-stream-output-stream` *echo-stream*) функция

Возвращает поток вывода для *echo-stream*.

(`file-length` *stream*) функция

Возвращает количество элементов в потоке *stream* или `nil`, если значение не может быть определено.

(`file-position` *stream* `&optional` *pos*) функция

Если *pos* не задан, возвращает текущее положение в потоке *stream* или `nil`, если положение не может быть определено. Если *pos* задан, он может быть `:start`, `:end` или неотрицательным целым числом, в соответствии с которым будет установлено новое положение в потоке *stream*.

(`file-string-length` *stream object*) функция

Возвращает разницу между текущим положением в потоке *stream* и положением после записи в него объекта *object*. Если разница не может быть определена, возвращает `nil`.

(`finish-output &optional` *stream*) функция

Принудительно выводит содержимое буфера выходного потока *stream*, затем возвращает `nil`.

(`force-output &optional` *stream*) функция

Действует подобно `finish-output`, но не ожидает завершения операции записи перед возвращением.

(`fresh-line &optional` *stream*) функция

Записывает перевод строки в поток *stream*, если текущее положение в потоке не соответствует началу новой строки.

(`get-output-stream-string` *stream*) функция

Возвращает строку, содержащую все знаки, переданные в поток *stream* (который должен быть открытым) с момента его открытия или последнего вызова `get-output-stream-string`.

(`input-stream-p` *stream*) функция

Возвращает истину, если *stream* является входным потоком.

(interactive-stream-p *stream*) функция

Возвращает истину, если *stream* является интерактивным потоком.

(listen &optional *stream*) функция

Возвращает истину, если имеются знаки, ожидающие прочтения из потока *stream*, который должен быть интерактивным.

(make-broadcast-stream &rest *streams*) функция

Возвращает новый широковещательный поток, составленный из *streams*.

(make-concatenated-stream &rest *input-streams*) функция

Возвращает новый конкатенированный поток, составленный из *input-streams*.

(make-echo-stream *input-stream output-stream*) функция

Возвращает новый эхо-поток, получающий ввод из *input-stream* и направляющий его в *output-stream*.

(make-string-input-stream *string* &optional *start end*) функция

Возвращает входной поток, который при чтении из него выдаст знаки, содержащиеся в строке *string*, после чего выдаст конец файла. Параметры *start* и *end* используются таким же образом, как и в других строковых функциях.

(make-string-output-stream &key *element-type*) функция

Возвращает выходной поток, принимающий знаки типа *element-type*. Записанные в него знаки никуда не перенаправляются, но могут быть считаны с помощью get-output-stream-string.

(make-synonym-stream *symbol*) функция

Возвращает поток, который будет синонимичен потоку, соответствующему специальной переменной с именем *symbol*.

(make-two-way-stream *input-stream output-stream*) функция

Возвращает новый двунаправленный поток, получающий ввод из *input-stream* и направляющий вывод в *output-stream*.

(open *path* &key *direction element-type if-exists if-does-not-exist external-format*) функция

Открывает и возвращает поток, связанный с *path*, или nil, если поток не может быть создан. Использует следующие аргументы по ключу:

:direction *symbol*

Определяет направление потока объектов. Может быть следующим: :input (по умолчанию) для чтения из потока, :output для записи в поток, :io для обеих операций или :probe, означающий, что поток будет возвращен закрытым.

:element-type *type*

Декларирует тип объектов, с которыми будет работать поток. Поток знаков должен использовать какой-либо подтип character; бинарный поток должен использовать подтип либо integer, либо signed-byte, либо unsigned-byte (в этом случае размер элементов будет определяться операционной системой). По умолчанию тип задан как character.

:if-exists *symbol*

Определяет действия в случае, когда указанный файл уже существует. Возможны следующие значения: :new-version, используемое по умолчанию, если компонента *version* файла *path* равна :newest, в противном случае по умолчанию используется :error; :rename, переименовывающее существующий файл; :rename-and-delete, переименовывающее и удаляющее существующий файл; :overwrite, выполняющее запись поверх существующих данных с начала файла; :append, выполняющее запись в существующий файл, добавляя новые знаки в конец; :supersede, заменяющее существующий файл новым с тем же именем (но исходный файл, вероятно, не удаляется до момента закрытия потока); nil, не создающий никаких потоков (open возвратит nil).

:if-does-not-exist *symbol*

Определяет действия в случае, когда указанный файл не найден. Возможны следующие значения: :error, используемое по умолчанию, если направление *direction* равно :input, или если :*if-exists* равен :overwrite или :append; :create, используемое по умолчанию, если направление *direction* равно :output или :io, или если *if-exists* не является ни :overwrite, ни :append; nil, используемое по умолчанию, если направление *direction* равно :probe (поток не создается, open возвращает nil).

:external-format *format*

Назначает формат файла. Единственным предопределенным форматом *format* является :default.

(open-stream-p *stream*) функция

Возвращает истину, если поток *stream* является открытым.

(output-stream-p *stream*) функция

Возвращает истину, если поток *stream* является потоком вывода.

(peek-char &optional *kind stream eof-error eof-value recursive*) функция

Возвращает текущий знак из потока *stream*, но не извлекает его из потока. Если *kind* – nil, то возвращается первый найденный знак; если t, то функция пропускает все пробельные символы и возвращает первый печатный знак; если же задан знак, то пропускает все зна-

ки, кроме заданного, и затем возвращает его. Если чтение дошло до конца файла, то либо вызывает ошибку, либо возвращается *eof-value* в зависимости от значения *eof-error* (истина или ложь; по умолчанию – истина). Параметр *recursive* будет истинным, если `peek-char` вызывается из другой считывающей функции.

(`read-byte` *stream* `&optional` *eof-error eof-value*) функция

Читает байт из бинарного входного потока *stream*. Если поток пуст, вызывается ошибка либо возвращается *eof-value* (в зависимости от *eof-error*).

(`read-char &optional` *stream eof-error eof-value recursive*) функция

Извлекает и возвращает первый знак из потока *stream*. Если встречен конец файла, сигнализирует об ошибке либо возвращает *eof-value* в зависимости от значения *eof-error* (по умолчанию – истина). Параметр *recursive* будет истинным, когда `read-char` вызывается из другой считывающей функции.

(`read-char-no-hang &optional` *stream eof-error eof-value recursive*) функция

Действует подобно `read-char`, но сразу же возвращает `nil`, если поток *stream* не содержит непрочтенных знаков.

(`read-line &optional` *stream eof-error eof-value recursive*) функция

Возвращает строку из знаков в потоке вплоть до первого знака переноса строки (он считывается, но не включается в строку результата) или знака конца файла. Если до знака конца файла нет ни одного другого знака, вызывается ошибка либо возвращаетя *eof-value* в зависимости от значения *eof-error* (по умолчанию – истина). Параметр *recursive* будет истинным, если `read-line` вызывается из другой считывающей функции.

(`read-sequence` ⟨*proseq*⟩ `&optional` *stream* `&key` *start end*) функция

Считывает элементы из *stream* в *proseq*, возвращая положение первого неизмененного элемента. Параметры *start* и *end* те же, что и в других строковых функциях.

(`stream-element-type` *stream*) функция

Возвращает тип объектов, которые могут быть прочитаны или записаны в *stream*.

(`stream-error-stream` *condition*) функция

Возвращает поток, связанный с условием *condition*.

(`stream-external-format` *stream*) функция

Возвращает внешний формат потока *stream*.

(`streamp` *object*) функция

Возвращает истину, если *object* является потоком.

(synonym-stream-symbol *synonym-stream*) функция

Возвращает имя специальной переменной, для которой значение *synonym-stream* является синонимом.

(terpri &optional *stream*) функция

Записывает перенос строки в *stream*.

(two-way-stream-input-stream *two-way-stream*) функция

Возвращает поток ввода для *two-way-stream*.

(two-way-stream-output-stream *two-way-stream*) функция

Возвращает поток вывода для *two-way-stream*.

(unread-char *character* &optional *stream*) функция

Отменяет одно прочтение read-char для *stream*. Не может применяться два раза подряд без использования read-char в промежутке между ними. Не может использоваться после peek-char.

(with-input-from-string (*symbol string* &key *index start end*) *declaration* expression**) макрос

Вычисляет выражения *expression*, связав *symbol* со строкой, полученной из потока ввода с помощью make-string-input-stream, вызванной с аргументами *string*, *start* и *end*. (Присваивание нового значения этой переменной внутри этой формы запрещено.) Создаваемый поток существует только внутри выражения with-input-from-string и закрывается автоматически после возврата значения with-input-from-string или прерывания его вычисления. Параметр *index* может быть выражением (не вычисляется), служащим первым аргументом setf; если вызов with-input-from-string завершается нормально, значение соответствующего места будет установлено в индекс первого непрочитанного из строки знака.

(with-open-file (*symbol path arg**) *declaration* expression**) макрос

Вычисляет выражения *expression*, связав *symbol* с потоком, полученным передачей пути *path* и параметров *arg* функции open. (Присваивание нового значения этой переменной внутри этой формы запрещено.) Поток существует только внутри выражения with-open-file и закрывается автоматически при возврате из выражения with-open-file или в случае прерывания его вычисления. В последнем случае, если поток открывался на запись, созданный файл удаляется.

(with-open-stream (*symbol stream*) *declaration* expression**) макрос

Вычисляет выражения, связав *symbol* со значением *stream*. (Присваивание нового значения этой переменной внутри формы запрещено.) Поток существует только внутри выражения with-open-stream и закрывается автоматически при возврате или прерывании with-open-stream.

(with-output-to-string (*symbol* [*string*] &key *element-type*) *declaration* expression**) макрос

Вычисляет выражения *expression*, связав *symbol* с потоком вывода для строки. Поток существует только внутри выражения with-output-to-string и закрывается при его возврате или прерывании. Если строка *string* не задана, with-output-to-string возвращает строку, состоящую из всего вывода, записанного в этот поток. Если строка *string* задается, она должна иметь указатель заполнения; знаки, передаваемые в поток, добавляются в конец строки так же, как при использовании vector-push-extend, а выражение with-output-to-string возвращает значение последнего выражения.

(write-byte *i stream*) функция

Записывает *i* в бинарный поток вывода *stream*. Возвращает *i*.

(write-char *character* &optional *stream*) функция

Записывает один знак *character* в поток *stream*.

(write-line *string* &optional *stream* &key *start end*) функция

Действует подобно write-string, но добавляет в конце перевод строки.

(write-sequence *proseq* &optional *stream* &key *start end*) функция

Записывает элементы последовательности *proseq* в поток *stream*. Параметры *start* и *end* те же, что и в других строковых функциях.

(write-string *string* &optional *stream* &key *start end*) функция

Записывает элементы строки *string* с поток *stream*. Параметры *start* и *end* те же, что и в других строковых функциях.

(yes-or-no-p &optional *format* &rest *args*) функция

Действует подобно y-or-n-p, но требует явного указания слов yes или no вместо одиночных знаков.

(y-or-n-p &optional *format* &rest *args*) функция

Отображает вопрос (да/нет) *format* (по умолчанию "") в *query-io* так, как это сделал бы вызов format с аргументами *args*. Затем выдает приглашение и ожидает ввода y или n, возвращая t или nil в зависимости от ответа. В случае некорректного ответа повторяет приглашение.

Печать

(copy-pprint-dispatch &optional *pprint-dispatch*) функция

Возвращает копию таблицы управления красивой печатью *pprint-dispatch*, которая по умолчанию – *print-pprint-dispatch*. Если *pprint-dispatch* – nil, то копирует исходное значение *print-pprint-dispatch*.

(format *dest format* &rest *args*) функция

Записывает вывод в *dest*: это либо поток, либо t (в этом случае запись происходит в *standard-output*), либо nil (запись происходит в строку, которая возвращается). Возвращает nil, если не была возвращена строка. Параметр *format* должен быть либо строкой, либо функцией, возвращаемой formatter. Если это функция, она применяется к *dest* и *args*. Если это строка, она может содержать директивы форматирования, обычно начинающиеся со знака ~, за которым следуют префиксные параметры, разделяемые запятыми, за которыми следуют необязательные модификаторы : и @, после которых следует определенная метка. Префиксом может быть: целое число; знак, предваряемый кавычкой; V или v, представляющий следующий аргумент (или отсутствие параметра, если аргумент – nil); #, представляющий количество оставшихся аргументов. Префиксные параметры могут пропускаться (запятые остаются), и в этом случае используется значение по умолчанию. Завершающие запятые также могут опускаться.

Допустимы следующие директивы:

~*w*, *g*, *m*, *p*A

Подобно princ, печатает следующий аргумент, сдвинутый вправо (или влево с помощью @) по меньшей мере на *m* (по умолчанию – 0) элементов *p* (по умолчанию – #\Space), а также элементы *p*, собранные в группы по *g* (по умолчанию – 1), до тех пор, пока суммарное количество знаков (включая представление аргумента) не будет больше или равно *w* (по умолчанию – 0). При использовании : пустые списки будут печататься как () вместо nil.

~*w*, *g*, *m*, *p*S

Похожа на ~A, но печатает аргумент, как prin1.

~W

Печатает следующий аргумент, как write. С модификатором : используется красивая печать, с @ не учитываются ограничения на длину и вложенность списка.

~C

Следующий аргумент должен быть знаком. Если это простой знак, он будет напечатан, как с помощью write-char. С модификатором : непечатные знаки убираются; :@ делает то же самое, но еще добавляет правила поведения для необычных знаков. С модификатором @ знаки печатаются в синтаксисе #\.

~*n*%

Выводит *n* (по умолчанию – 1) переводов строки.

~*n*&

Похожа на ~%, но первый перевод строки выполняется так же, как с помощью fresh-line.

~*n*|

Печатает *n* (по умолчанию – 1) разделителей страниц.

~*n*~

Печатает *n* (по умолчанию – 1) ~s.

~*r*, *w*, *p*, *c*, *i*R

Следующий аргумент должен быть целым числом. Он печатается в системе счисления с основанием *r* (по умолчанию – 10) и со смещением влево на столько *p* (по умолчанию – #\Space), сколько требуется для того, чтобы суммарное количество знаков было равно, по меньшей мере, *w*. С модификатором : группы из *i* (по умолчанию – 3) знаков разделяются элементами *c* (по умолчанию – #\,). С модификатором @ знак печатается даже для положительных чисел.

Если никакие префиксы не предоставляются, ~R имеет совершенно другое значение, отображая числа в разнообразных нечисловых форматах. Без модификаторов 4 будет напечатано как four, с модификатором : – как fourth, с @ – как IV, а с :@ – как IIII.

~*w*, *p*, *c*, *i*D

Отображает число в десятичном формате. Эквивалент ~10, *w*, *p*, *c*, *i*R.

~*w*, *p*, *c*, *i*B

Отображает число в бинарном формате. Эквивалент ~2, *w*, *p*, *c*, *i*R.

~*w*, *p*, *c*, *i*O

Отображает число в восьмеричном формате. Эквивалент ~8, *w*, *p*, *c*, *i*R.

~*w*, *p*, *c*, *i*X

Отображает числа в шестнадцатеричном формате. Эквивалент ~16, *w*, *p*, *c*, *i*R.

~*w*, *d*, *s*, *x*, *p*F

Если следующий аргумент является рациональной дробью, он печатается как десятичная дробь, смещенная на *s* цифр влево (по умолчанию – 0) с помощью *d* цифр (по умолчанию столько, сколько потребуется) после десятичного разделителя. Число может округляться, но в какую сторону – зависит от реализации. Если задано *w*, число будет сдвинуто влево на столько *p* (по умолчанию – #\Space), сколько потребуется, чтобы полное количество знаков было равно *w*. (Если само представление числа содержит больше знаков, чем *w*, и задан *x*, то он будет напечатан вместо каждого другого знака.) Если опущены *w* и *d*, то *s* игнорируется. С модификатором @ знак печатается даже для положительных чисел.

~*w*, *d*, *e*, *s*, *x*, *p*, *m*E

То же, что и ~F, но если следующий аргумент – рациональная дробь, он печатается в экспоненциальной записи с *s* (по умолчанию – 1)

знаками перед десятичным разделителем, *d* (по умолчанию столько, сколько потребуется) знаками после него и *e* (по умолчанию столько, сколько потребуется) знаками в экспоненте. Если задан *m*, он заменит знак экспоненты.

~*w*, *d*, *e*, *s*, *x*, *p*, *m*G

Если следующий аргумент является рациональной дробью, он будет отображен с помощью ~F или ~E в зависимости от числа.

~*d*, *n*, *w*, *p*$

Используется для печати денежных сумм. Если следующий аргумент является рациональной дробью, он будет напечатан в десятичном представлении с, по меньшей мере, *n* (по умолчанию – 1) цифрами перед десятичным разделителем и *d* цифрами (по умолчанию – 2) после него. Будет напечатано, по меньшей мере, *w* (по умолчанию – 0) знаков; если потребуется, то число будет смещено влево на *p* (по умолчанию – #\Space). С модификатором @ знак будет напечатан даже перед положительными числами. С модификатором : знак будет напечатан перед смещением.

Подобным образом действует pprint-newline с аргументами, зависящими от модификаторов: отсутствие соответствует :linear, @ соответствует :miser, : соответствует :fill, а :@ соответствует :mandatory.

~<*prefix*~;*body*~;*suffix*~:>

Действует подобно pprint-logial-block, требуя, чтобы следующий аргумент был списком. *prefix* и *suffix* аналогичны :prefix (или :per-line-prefix, если за ним следует ~@;) и :suffix, а *body* играет роль выражений, составляющих тело. *body* может быть любой форматирующей строкой, а аргументы для нее получаются из списка аргументов, как при использовании pprint-pop. Внутри тела ~^ соответствует pprint-exit-if-list-exhausted. Если из трех параметров *prefix*, *body* и *suffix* заданы лишь два, то значением *suffix* по умолчанию считается ""; если задан лишь один из трех, то *prefix* также считается "". С модификатором : *prefix* и *suffix* имеют значения по умолчанию "(" и ")" соответственно. С модификатором @ список оставшихся аргументов становится аргументом логического блока. Если вся директива заканчивается ~:@>, то новая условная строка :fill добавляется после каждой группы пробелов в *body*.

~*n*I

Эквивалент (pprint-indent :block *n*). С модификатором : аналогично (pprint-indent :current *n*).

~/*name*/

Вызывает функцию с именем *name* и, по меньшей мере, четырьмя аргументами: поток; следующий аргумент; два значения, истинных, когда используется : или @ соответственно, а также любыми другими параметрами, переданными директиве.

~*m*, *n*T

Печатает достаточное количество пробелов, чтобы переместить курсор в колонку *m* (по умолчанию – 1), а если данная колонка уже пройдена, то в ближайшую колонку с несколькими интервалами *n* (по умолчанию – 1) после *m*. С модификатором @ печатает *m* пробелов, затем достаточное количество пробелов, кратное *n*. С модификатором : эквивалентна (pprint-tab :section *m n*). С @ эквивалентна (pprint-tab :section-relative *m n*).

~*w*, *n*, *m*, *p*<+*text0*~;...~;*textn*~>

Отображает знаки, полученные смещением значений *text* на поле шириной *w* знаков, с, по меньшей мере, *m* (по умолчанию – 0) дополнительными элементами *p* (по умолчанию – #\Space), при необходимости вставляемыми между *text*. Если ширина вывода с учетом минимального отступа больше *w*, ограничение увеличивается на число, кратное *n*. С модификатором : отступ будет также перед первым текстом; с @ – после последнего текста. ~^ завершает обработку директивы.

Если *text0* следует за ~*a*,*b*:;, а не ~;, знаки, полученные из *text0*, будут выводиться при условии, что оставшихся знаков больше, чем *b* (по умолчанию – длина строки потока или 72) с учетом *a* (по умолчанию – 0) зарезервированных знаков.

~*n**

Игнорирует следующие *n* (по умолчанию – 0) аргументов. С модификатором : сохраняет *n* аргументов. С @ делает *n*-й аргумент (индексация с нуля) текущим.

~*i*[*text0*~;...~; *textn*~]

Отображает текст, полученный из *texti*. Если *i* не задан, используется значение следующего аргумента. Если последнему *text* предшествовала ~:;, а не ~;, то будет использоваться последнее значение *text* (если никакое другое не было выбрано). С модификатором : предполагается наличие только двух *text*: второй используется, если следующий аргумент истинен, первый – в противном случае. С @ предполагается наличие только одного *text*: если следующий аргумент истинен, выводятся знаки текста и аргумент сохраняется для следующей директивы.

~*n*{*text*~}

Действует подобно повторным вызовам format с *text* в качестве форматирующей строки и параметрами, взятыми из следующего аргумента, являющегося списком. Выполняется до тех пор, пока не закончатся аргументы, или при достижении *n*-го аргумента (по умолчанию – без ограничений) начинает сначала. Если директива завершается ~:} вместо ~}, то будет сделан по крайней мере один вызов format, пока *n* не будет равным 0. Если *text* не задан, вместо него используется очередной аргумент (должен быть строкой). ~^ завер-

шает обработку директивы. С : аргумент должен быть списком списков, а их элементы будут аргументами последовательных вызовов format. С @ вместо очередного аргумента будет использоваться список оставшихся аргументов.

~?

Рекурсивный вызов format: очередной аргумент воспринимается как строка форматирования, а следующие за ним аргументы (в нужном количестве) – как аргументы для вызова format с этой строкой. С @ очередной аргумент будет использоваться как строка форматирования, но соответствующие параметры будут браться из аргументов основного вызова format.

~(*text*~)

Печатает *text*, преобразуя регистр в зависимости от модификатора: с : первая буква каждого слова преобразуется к верхнему регистру; с @ помимо этого все остальные буквы преобразуются к нижнему регистру; с :@ все буквы преобразуются к верхнему регистру.

~P

Если первый аргумент равен (eql) 1, не печатает ничего, иначе печатает s. С : предварительно сохраняет первый аргумент. С @, если первый аргумент равен 1, печатает y, иначе – ies; с :@ предварительно сохраняет один аргумент.

~*a*, *b*, *c*^

Завершает директиву format (или текущую директиву, если используется внутри нее) при следующих условиях: если не заданы никакие префиксные параметры; если задан один параметр и он равен 0; если заданы два равных параметра; если заданы три параметра и выполняется (<= *a b c*).

Если за ~ следует перевод строки, то перевод строки и все следующие за ним пробелы игнорируются. С @ игнорируется перевод строки, но не пробелы.

(formatter *string*) макрос

Возвращает функцию, принимающую поток и ряд других параметров, и применяет format к потоку, форматирующей строке и другим параметрам, возвращая лишние параметры.

(pprint *object* &optional *stream*) функция

Действует подобно print, но пытается красивее расставить отступы, а также не печатает висячие пробелы.

(pprint-dispatch *object* &optional *pprint-dispatch*) функция

Возвращает наиболее приоритетную функцию в *pprint-dispatch*, для которой *object* имеет ассоциированный тип. Если *pprint-dispatch* не передан, используется текущее значение *print-pprint-dispatch*; если

передан nil, используется исходное значение. Если ни один тип в таблице диспетчеризации не соответствует *object*, возвращается функция, печатающая его с помощью print-object. Второе возвращаемое значение истинно, когда первое было получено из таблицы диспетчеризации.

(pprint-exit-if-list-exhausted) функция

Употребляется вместе с pprint-logical-block. Завершает блок, если больше нечего печатать, иначе возвращает nil.

(pprint-fill *stream object* &optional *colon at*) функция

Печатает объект *object* в поток *stream*, но делает это иначе, если это список и значение *print-pretty* истинно. В случае истинности *colon* (по умолчанию) печатает столько элементов списка, сколько способно уместиться в строке, окруженной скобками. Аргумент *at* игнорируется. Возвращает nil.

(pprint-indent *keyword r* &optional *stream*) функция

Если *stream* был создан с помощью pprint-logical-block и значение *print-pretty* истинно, устанавливает отступ для текущего логического блока. Если ключ *keyword* равен :current, отступ устанавливается в текущую позицию, смещенную на *r*; если ключ установлен в :block, позиция отсчитывается от первого знака в текущем блоке, к которому добавляется *r*.

(pprint-linear *stream object* &optional *colon at*) функция

Действует подобно pprint-fill, но умещает весь список на одной строке или каждый элемент на отдельной строке.

(pprint-logical-block (*symbol object* макрос
&key *prefix per-line-prefix suffix*)
declaration expression**)

Вычисляет выражения *expressions*, связав *symbol* с новым потоком (допустимым внутри выражения pprint-logical-block), который направляет вывод в исходное значение *symbol* (которое должно быть потоком). Весь вывод, направляемый в новый поток, находится в связанном с ним логическом блоке. Выражения *expression* не должны иметь побочных эффектов в данном окружении.

Объект *object* должен быть списком, который будут печатать выражения *expressions*. Если это не список, он будет напечатан с помощью write; если список, то его элементы будут получаться по очереди вызовом pprint-pop. При печати списка его разделяемые и вложенные компоненты будут отображаться согласно *print-circle* и *print-level*.

Аргументы по ключу, если заданы, должны вычисляться в строки. *prefix* будет напечатан перед логическим блоком, *suffix* – после него, а *per-line-prefix* – в начале каждой строки. Параметры *prefix* и *per-line-prefix* являются взаимоисключающими.

(pprint-newline *keyword* &optional *stream*) функция

Если поток *stream* был создан с помощью pprint-logical-block и значение *print-pretty* истинно, записывает новую строку в поток *stream* с учетом *keyword*: :mandatory означает всегда; :linear – если результат красивой печати не умещается на строке; :miser – для экономной записи; :fill – если предыдущая или следующая строки будут разорваны.

(pprint-pop) макрос

Используется внутри pprint-logical-block. Если список элементов, печатаемых в текущем логическом блоке, не пуст, возвращает следующий элемент. Если в списке остался атом (не nil), печатает его вслед за точкой и возвращает nil. Если значение *print-length* истинно и соответствует количеству уже напечатанных элементов, печатает многоточие и возвращает nil. Если значение *print-circle* истинно и остаток списка является разделяемой структурой, печатает точку, за которой следует #n#, и возвращает nil.

(pprint-tab *keyword i1 i2* &optional *stream*) функция

Если поток *stream* был создан с помощью pprint-logical-block и значение *print-pretty* истинно, выводится табуляция так же, как с помощью директивы форматирования ~T с префиксными параметрами *i1* и *i2*. Возможны следующие ключевые слова: :line соответствует ~T, :section – ~:T, :line-relative – ~@T, :section-relative – ~:@T.

(pprint-tabular *stream object* &optional *colon at tab*) функция

Действует подобно pprint-fill, но печатает элементы списка так, что они выстраиваются в столбцы. Параметр *tab* (по умолчанию – 16) соответствует расстоянию между столбцами.

(princ *object* &optional *stream*) функция

Отображает объект *object* в поток *stream* в наиболее удобном для восприятия виде (по возможности). Escape-знаки (экранирующие символы) не отображаются.

(princ-to-string *object*) функция

Действует подобно princ, но направляет вывод в строку, которую затем возвращает.

(print *object* &optional *stream*) функция

Действует подобно prin1, но окружает объект переносами строк.

(print-object *object stream*) обобщенная функция

Вызывается системой при печати объекта *object* в поток *stream*.

(print-not-readable-object *condition*) функция

Возвращает объект, который не удалось напечатать в читаемом виде при возникновении условия *condition*.

(print-unreadable-object (*object stream* &key *type identity*) *expression**) макрос

Используется для отображения объектов в синтаксисе #<...>. Все аргументы вычисляются. Записывает #< в поток *stream*; затем, если *type* является типом, выводит метку типа для объекта; затем вычисляет выражения *expression*, которые должны отображать объект *object* в поток *stream*; затем, если значение *identity* истинно, записывает идентификатор объекта; в конце записывает >. Возвращает nil.

(prin1 *object* &optional *stream*) функция

Отображает объект *object* в поток *stream* таким образом, чтобы он (по возможности) мог быть впоследствии прочитан с помощью read.

(prin1-to-string *object*) функция

Действует подобно prin1, но направляет вывод в строку, которую затем возвращает.

(set-pprint-dispatch *type function* &optional *r pprint-dispatch*) функция

Если значение *function* истинно, добавляет элемент в *pprint-dispatch* (по умолчанию *print-pprint-dispatch*) с заданным типом *type*, функцией *function* и приоритетом *r* (по умолчанию – 0). Функция *function* должна принимать два аргумента: поток и объект для печати. Если *function* – nil, то удаляет из таблицы все элементы с типом *type*. Возвращает nil.

(write *object* &key *array base case circle escape gensym length level lines miser-width pprint-dispatch pretty radix readably right-margin stream*) функция

Записывает объект *object* в поток *stream*, связывая каждую специальную переменную вида *print-...* со значением соответствующего аргумента по ключу.

(write-to-string *object* &key ...) функция

Действует подобно write, но перенаправляет вывод в строку, которую затем возвращает.

Считыватель

(copy-readtable &optional *from* ⟨*to*⟩) функция

Если *to* – nil, возвращает копию таблицы чтения (по умолчанию – *readtable*); если *to* – таблица чтения, возвращает ее после копирования в нее таблицы *from*.

(get-dispatch-macro-character *char1 char2* &optional *readtable*) функция

Возвращает функцию (или nil, если ничего не найдено), вызываемую в *readtable*, если во вводе *char1* следует за *char2*.

(get-macro-character *char* &optional *readtable*) функция

Возвращает два значения: функцию (или nil, если ничего не найдено), вызываемую в *readtable*, когда во вводе встречается *char*, а также истину в качестве второго значения, если *char* был прочитан как часть имени символа.

(make-dispatch-macro-character *char* &optional *nonterm readtable*) функция

Добавляет в *readtable* управляющий макрознак *char*. Если значение *nonterm* истинно, *char*, считанный в середине символа, ведет себя как нормальный знак. Возвращает t.

(read &optional *stream eof-error eof-value recursive*) функция

Считывает один Лисп-объект из потока *stream*, обрабатывает и возвращает его. Если достигнут конец файла, либо сигнализируется ошибка, либо возвращается *eof-value* в зависимости от значения *eof-error* (по умолчанию – истина). При вызове read из другой считывающей функции будет передан истинный параметр *recursive*.

(read-delimited-list *char* &optional *stream recursive*) функция

Действует подобно read, но продолжает обработку объектов из потока *stream* до тех пор, пока не встретит *char*, после чего немедленно возвращает список обработанных объектов. Сигнализирует ошибку, если до конца файла не остается ни одного знака *char*.

(read-from-string *string* &optional *eof-error eof-value* &key *start end preserve-whitespace*) функция

Действует по отношению к строке *string* так же, как read к потоку. Возвращает два значения: обработанный объект и положение (индексация начинается с нуля) первого непрочитанного знака в строке *string*. Если значение *preserve-whitespace* истинно, действует как read-preserving-whitespace вместо read.

(read-preserving-whitespace &optional *stream eof-error eof-value recursive*) функция

Действует подобно read, но оставляет в потоке завершающие пробелы.

(readtable-case *readtable*) функция

Возвращает один из вариантов: :upcase, :downcase, :preserve или :invert в зависимости от способа работы *readtable* с регистрами во вводе. Может быть первым аргументом setf.

(readtablep *object*) функция

Возвращает истину, если объект *object* является таблицей чтения.

(set-dispatch-macro-character *char1 char2 function* &optional *readtable*) функция

Добавляет в *readtable* элемент, сообщающий, что *function* будет вызываться при последовательном считывании *char1* и *char2* (который

преобразуется к верхнему регистру и не может быть цифрой). *function* должна быть функцией трех аргументов – входной поток, *char1* и *char2* – и возвращать объект, прочитанный из потока, либо не возвращать ничего. Возвращает `t`.

(`set-macro-character` *char function* `&optional` *nonterm readtable*) функция

Добавляет в *readtable* элемент, сообщающий, что *function* будет вызываться при считывании *char*. *function* должна быть функцией двух аргументов – входной поток и *char* – и возвращать либо считанный объект, либо не возвращать ничего. Если значение *nonterm* истинно, *char*, считанный в середине символа, ведет себя как нормальный знак. Возвращает `t`.

(`set-syntax-from-char` *to-char from-char to-readtable from-readtable*) функция

Передает знаку *to-char* в таблице *to-readtable* (по умолчанию – `*readtable*`) синтаксические свойства знака *from-char* из таблицы *from-readtable* (по умолчанию – стандартная таблица). Возвращает `t`.

(`with-standard-io-syntax` *expression**) макрос

Вычисляет выражения *expression*, связав все специальные переменные, управляющие чтением и печатью (т. е. с именами вида `*read-` или `*print-`), с их исходными значениями.

Сборка системы

(`compile-file` *path* `&key` *output-file verbose print external-format*) функция

Компилирует содержимое файла, на который указывает *path*, и записывает результат в файл *output-file*. Если *verbose* истинен, факт компиляции записывается в `*standard-output*`. Если *print* истинен, информация о toplevel-выражениях выводится в `*standard-output*`. Единственным (по стандарту) форматом *external-format* является `:default`. После завершения компиляции устанавливаются исходные значения `*readtable*` и `*package*`. Возвращает три значения: имя выходного файла (или `nil`, если файл не смог записаться); значение, истинное, если в результате компиляции были получены ошибки или предупреждения; еще одно значение, истинное, если были получены ошибки или предупреждения, отличные от стилевых предупреждений.

(`compile-file-pathname` *path* `&key` *output-file*) функция

Возвращает имя выходного файла, который создаст `compile-file`, вызванная с теми же аргументами. Допускает использование других ключей.

(load *path* &key *verbose print if-does-not-exist external-format*) функция

Загружает файл, на который указывает *path*. Если это файл с исходным кодом, то выполняется последовательное вычисление всех toplevel-выражений. То же самое происходит и для скомпилированного файла, но при этом для соответствующих функций из него справедлив предикат compiled-function-p. Если параметр *verbose* истинен, информация о загрузке файла анонсируется в *standard-output*. Если параметр *print* истинен, процесс загрузки также выводится в *standard-output*. Если параметр *if-does-not-exist* истинен (по умолчанию), в случае отсутствия файла сигнализируется ошибка; в противном случае возвращается nil. Единственным (по стандарту) форматом *external-format* является :default. Возвращает t, если файл загружен.

(provide *name*) [функция]

Добавляет строку *name* (или имя соответствующего символа) в *modules*.

(require *name* &optional *paths*) [функция]

Если строка *name* (или имя соответствующего символа) не находится в *modules*, пытается загрузить файл, содержащий соответствующий модуль. Параметр *path*, если задается, должен быть списком путей, потоков или строк, при этом будут загружаться указанные файлы.

(with-compilation-unit ([:override *val*]) *expression**) макрос

Вычисляет выражения *expression*. Вывод предупреждений, задерживаемых до конца компиляции, будет задержан до завершения вычисления последнего выражения. В случае вложенного вызова эффект будет получен только при сообщении истинного значения *val*.

Окружение

(apropos *name* &optional *package*) функция

Выводит информацию о каждом интернированном символе, имя которого содержит подстроку *name* (или имя *name*, если это символ). Если задан пакет *package*, поиск осуществляется только по нему. Не возвращает значений.

(apropos-list *name* &optional *package*) функция

Действует подобно apropos, но не выводит информацию, а возвращает список символов.

(decode-universal-time *i* &optional *time-zone*) функция

Интерпретирует *i* как количество секунд с момента 1 января 1990 года, 00:00:00 (GMT), возвращая девять значений: секунды, минуты, час, день, месяц (январю соответствует 1), год, день недели (понедельнику соответствует 0), действует ли летнее время и рациональное число, соответствующее смещению временной зоны относитель-

но GMT. Параметр *time-zone* должен быть рациональным числом в интервале от -24 до 24 включительно; если он задается, летнее время не учитывается.

(describe *object* &optional *stream*) функция

Записывает описание объекта *object* в поток *stream*. Не возвращает значений.

(describe-object *object* &optional *stream*) обобщенная функция

Вызывается из describe для описания объекта *object* из потока *stream*.

(disassemble *fn*) функция

Выводит объектный код, сгенерированный для *fn*, которая может быть функцией, именем функции или лямбда-выражением.

(documentation *object symbol*) обобщенная функция

Возвращает документацию для *object* с ключом *symbol* или nil, если ничего не найдено. Может устанавливаться с помощью setf.

(dribble &optional *path*) функция

Если задан путь *path*, начинает пересылать в соответствующий файл вывод данной Лисп-сессии. Если вызывается без аргументов, то закрывает данный файл.

(ed &optional *arg*) функция

Вызывает редактор (если имеется хотя бы один). Если *arg* – путь или строка, то будет редактироваться соответствующий файл. Если это имя функции, редактироваться будет ее определение.

(encode-universal-time *second minute hour date month year* &optional *time-zone*) функция

Функция, обратная decode-unversal-time.

(get-decoded-time) функция

Эквивалент (decode-universal-time (get-universal-time)).

(get-internal-real-time) функция

Возвращает текущее системное время в тактах системных часов, которые составляют internal-time-units-per-second долю секунды.

(get-internal-run-time) функция

Действует подобно get-internal-real-time, но возвращает значение, соответствующее чистому времени работы Лисп-процесса в системных тактах.

(get-universal-time) функция

Возвращает текущее время как количество секунд с 1 января 1900 года, 00:00:00 (GMT).

(inspect *object*) функция

Интерактивная версия describe, позволяющая перемещаться по составным объектам.

(lisp-implementation-type) функция

Возвращает строку, характеризующую используемую реализацию, или nil.

(lisp-implementation-version) функция

Возвращает строку, характеризующую версию используемой реализации, или nil.

(long-site-name) функция

Подобна short-site-name, но выводит больше информации.

(machine-instance) функция

Возвращает строку, характеризующую конкретную машину, на которой выполняется Лисп-сессия, или nil.

(machine-type) функция

Возвращает строку, характеризующую тип машины, на которой выполняется Лисп-сессия, или nil.

(machine-version) функция

Возвращает строку, сообщающую версию машины, на которой выполняется Лисп-сессия, или nil.

(room &optional (*arg* :default)) функция

Выводит сообщение, описывающее текущее состояние памяти. Более лаконичный ответ будет получен с аргументом nil, более многословный – с t.

(short-site-name) функция

Возвращает строку, характеризующую текущее физическое расположение, или nil.

(sleep *r*) функция

Приостанавливает вычисление на *r* секунд.

(software-type) функция

Возвращает строку, характеризующую тип нижележащего программного обеспечения, или nil.

(software-version) функция

Возвращает строку, характеризующую версию нижележащего программного обеспечения, или nil.

(step *expression*) макрос

Делает шаг в вычислении выражения *expression*, возвращая значение(я), которые будут получены.

(time *expression*) макрос

Вычисляет выражение *expression*, возвращая полученное значение(я), а также выводя в *trace-output* информацию о затраченном времени.

(trace *fname**) макрос

Выводит информацию о вызовах функций с заданными именами в *trace-output*. Может не работать для функций, скомпилированных с inline-декларацией. Вызванная без аргументов, выводит список функций, отслеживаемых на данный момент.

(untrace *fname**) макрос

Отменяет вызов trace. Вызванная без аргументов, отменяет трассировку всех отслеживаемых функций.

(user-homedir-pathname &optional *host*) функция

Возвращает путь к домашнему каталогу пользователя или nil, если на заданном хосте его нет. Параметр *host* используется так же, как в make-pathname.

Константы и переменные

array-dimension-limit константа

Положительный fixnum, на единицу превышающий ограничение на количество элементов в одной размерности массива. Зависит от реализации, но не менее 1024.

array-rank-limit константа

Положительный fixnum, на единицу превышающий максимальное количество размерностей в массиве. Зависит от реализации, но не менее 8.

array-total-size-limit константа

Положительный fixnum, на единицу превышающий ограничение на количество элементов в массиве. Зависит от реализации, но не менее 1024.

boole-1 ... boole-xor константы

Положительные целые числа, используемые как первый аргумент boole.

break-on-signals переменная

Обобщение более старой *break-on-warnings*. Значение переменной должно быть спецификатором типа. Если сигнализируется условие заданного типа, вызывается отладчик. Исходно равна nil.

`call-arguments-limit` константа

Положительное целое число, на единицу большее максимального количества аргументов в вызове функции. Зависит от реализации, но не менее 50.

`char-code-limit` константа

Положительное целое число, на единицу большее максимального значения, возвращаемого `char-code`. Зависит от реализации.

`*compile-file-pathname*` переменная

В процессе вызова `compile-file` содержит путь, полученный из первого аргумента. Вне процесса компиляции – `nil`.

`*compile-file-truename*` переменная

Настоящее имя для `*compile-file-pathname*`.

`*compile-print*` переменная

Используется как значение по умолчанию параметра `:print` в `compile-file`. Исходное значение зависит от реализации.

`*compile-verbose*` переменная

Используется как значение по умолчанию параметра `:verbose` в `compile-file`. Исходное значение зависит от реализации.

`*debug-io*` переменная

Поток, используемый для интерактивной отладки.

`*debugger-hook*` переменная

Если не является `nil`, то должна быть функцией *f* двух аргументов. Непосредственно перед вызовом отладчика *f* вызывается с возникшим условием и самой *f*. Если вызов *f* завершается нормально, управление передается в отладчик. В процессе вызова *f* `*debugger-hook*` будет установлен в `nil`.

`*default-pathname-defaults*` переменная

Используется как значение по умолчанию, когда функциям типа `make-pathname` не передан аргумент `:defaults`.

`short-float-epsilon` константа
`single-float-epsilon` константа
`double-float-epsilon` константа
`long-float-epsilon` константа

Для каждого соответствующего типа определяет наименьшее положительное число того же формата, которое при добавлении к нему `1.0` в том же формате приводит к результату, отличному от `1.0`. Зависит от реализации.

short-float-negative-epsilon константа
single-float-negative-epsilon константа
double-float-negative-epsilon константа
long-float-negative-epsilon константа

Для каждого соответствующего типа определяет наименьшее положительное число того же формата, которое при вычитании его из 1.0 в том же формате приводит к результату, отличному от 1.0. Зависит от реализации.

error-output переменная

Поток, используемый для вывода сообщений об ошибках.

features переменная

Зависящий от реализации список символов, представляющих возможности, поддерживаемые данной реализацией. Эти символы могут использоваться как тесты в #+ и #-.

gensym-counter переменная

Неотрицательное целое число, используемое gensym для создания имен символов. Исходное значение зависит от реализации.

internal-time-units-per-second константа

Если разница между двумя вызовами get-internal-run-time делится на данное целое число, результат будет представлен в виде количества секунд системного времени между ними.

lambda-list-keywords константа

Список ключей в списках параметров лямбда-выражения (например, &optional, &rest и т. д.), поддерживаемых реализацией.

lambda-parameters-limit константа

Положительное целое число, на единицу большее максимального количества переменных в списке параметров лямбда-выражения. Зависит от реализации, но не меньше 50.

least-negative-short-float константа
least-negative-single-float константа
least-negative-double-float константа
least-negative-long-float константа

Наименьшее по модулю отрицательное число с плавающей запятой каждого типа, поддерживаемое реализацией.

least-negative-normalized-short-float константа
least-negative-normalized-single-float константа
least-negative-normalized-double-float константа
least-negative-normalized-long-float константа

Наименьшее по модулю нормализованное отрицательное число с плавающей запятой каждого типа, поддерживаемое реализацией.

least-positive-short-float константа
least-positive-single-float константа
least-positive-double-float константа
least-positive-long-float константа

Наименьшее по модулю положительное число с плавающей запятой каждого типа, поддерживаемое реализацией.

least-positive-normalized-short-float константа
least-positive-normalized-single-float константа
least-positive-normalized-double-float константа
least-positive-normalized-long-float константа

Наименьшее по модулю нормализованное положительное число с плавающей запятой каждого типа, поддерживаемое реализацией.

load-pathname переменная

В процессе вычисления вызова load связывается с путем, полученным из первого аргумента. Вне процесса загрузки – nil.

load-print переменная

Используется как значение по умолчанию параметра :print в вызове load. Исходное значение зависит от реализации.

load-truename переменная

Настоящее имя для *load-pathname*.

load-verbose переменная

Используется как значение по умолчанию параметра :verbose в вызове load. Исходное значение зависит от реализации.

macroexpand-hook переменная

Функция трех аргументов – раскрывающая функция, макровызов и окружение – вызывается из macroexpand-1 для генерации раскрытий макросов. Исходное значение эквивалентно funcall или имени соответствующей функции.

modules переменная

Список строк, полученных в результате вызовов provide.

most-negative-fixnum константа

Наименьшее возможное значение типа fixnum, поддерживаемое реализацией.

most-negative-short-float константа
most-negative-single-float константа
most-negative-double-float константа
most-negative-long-float константа

Набольшее по модулю отрицательное число с плавающей запятой каждого типа, поддерживаемое реализацией.

most-positive-fixnum константа

Наибольшее возможное значение типа fixnum, поддерживаемое реализацией.

most-positive-short-float константа
most-positive-single-float константа
most-positive-double-float константа
most-positive-long-float константа

Набольшее по модулю положительное число с плавающей запятой каждого типа, поддерживаемое реализацией.

multiple-values-limit константа

Положительное целое число, на единицу большее максимального количества возвращаемых значений. Зависит от реализации, но не менее 20.

nil константа

Вычисляется сама в себя. В равной степени представляет ложь и пустой список.

package переменная

Текущий пакет. Исходно – common-lisp-user.

pi константа

Приближение числа π в формате long-float.

print-array переменная

Если истинна, массивы всегда печатаются в читаемом (readable) формате. Исходное значение зависит от реализации.

print-base переменная

Целое число между 2 и 36 включительно. Определяет основание системы счисления, в которой печатаются числа. Исходное значение равно 10 (десятичный формат).

print-case переменная

Управляет печатью символов, имена которых находятся в верхнем регистре. Возможны три значения: :upcase (исходное значение), :downcase и :capitalize (печатает символы так, как после применения string-capitalize к их именам).

print-circle переменная

Если истинна, разделяемые структуры будут отображаться с помощью макросов чтения #n= и #n#. Исходно равна nil.

print-escape переменная

Если равна nil, то все будет печататься так же, как с помощью princ. Исходно равна t.

`*print-gensym*` переменная

Если истинна, то перед неинтернированными символами будет печататься `#:`. Исходно равна `t`.

`*print-length*` переменная

Либо `nil` (исходно), либо положительное целое число, которое определяет максимальное количество элементов, отображаемых для одного объекта (лишние будут скрыты). Если равна `nil`, то ограничения нет.

`*print-level*` переменная

Либо `nil` (исходно), либо положительное целое число, которое определяет максимальный уровень вложенности отображаемых объектов (остальное будет скрыто). Если равна `nil`, то ограничения нет.

`*print-lines*` переменная

Либо `nil` (исходно), либо положительное целое число, которое определяет максимальное количество строк, используемых для отображения объекта при красивой печати (лишние будут скрыты). Если равна `nil`, то ограничения нет.

`*print-miser-width*` переменная

Либо `nil`, либо положительное целое число, которое задает компактный стиль красивой печати, используемый, если необходимо ужать представление объекта. Исходное значение зависит от реализации.

`*print-pprint-dispatch*` переменная

Либо `nil`, либо таблица управления красивой печатью. Исходно установлена таблица, выполняющая красивую печать согласно обычным правилам.

`*print-pretty*` переменная

Когда истинна, для отображения объектов используется красивая печать. Исходное значение зависит от реализации.

`*print-radix*` переменная

Когда истинна, задает основание системы счисления печатаемых чисел. Исходное значение — `nil`.

`*print-readably*` переменная

Когда истинна, принтер должен либо сгенерировать читаемый вывод, либо сигнализировать ошибку. Изначально установлена в `nil`.

`*print-right-margin*` переменная

Либо `nil` (исходно), либо положительное целое число, представляющее величину правого поля, на котором ничего не будет печататься. Если равна `nil`, величина правого поля задается самим потоком.

`*query-io*` переменная

Поток, используемый для двустороннего общения с пользователем.

`*random-state*` переменная

Объект, представляющий состояние генератора случайных чисел Common Lisp.

`*read-base*` переменная

Целое число от `2` до `36` включительно. Определяет основание системы счисления, в которой будут считываться числа. Исходно равна `10` (десятичный формат).

`*read-default-float-format*` переменная

Показывает формат, используемый `read` по умолчанию для чтения чисел с плавающей запятой. Должна быть спецификатором типа чисел с плавающей запятой. Исходное значение — `single-float`.

`*read-eval*` переменная

Если `nil`, `#.` сигнализирует ошибку. Исходное значение — `t`.

`*read-suppress*` переменная

Если истинна, `read` будет более толерантной к синтаксическим различиям. Исходно равна `nil`.

`*readtable*` переменная

Текущая таблица чтения. Исходная таблица задает стандартный синтаксис Common Lisp.

`*standard-input*` переменная

Входной поток по умолчанию.

`*standard-output*` переменная

Выходной поток по умолчанию.

`t` константа

Вычисляется сама в себя. Представляет истину.

`*terminal-io*` переменная

Поток, используемый консолью (если имеется).

`*trace-output*` переменная

Поток, в который отображается результат трассировки.

`* ** ***` переменные

Содержат первые значения соответственно трех последних выражений, введенных в toplevel.

`+ ++ +++` переменные

Содержат соответственно три последних выражения, введенных в toplevel.

- переменная

В процессе вычисления выражения в toplevel содержит само выражение.

/ // /// переменные

Содержат список значений, которые вернули соответственно три последних выражения, введенные в toplevel.

Спецификаторы типов

Спецификаторы типов могут быть простыми или составными. Простой спецификатор – это символ, соответствующий имени типа (например, `integer`). Составной спецификатор – это список из символа и набора аргументов. В этом разделе будут рассмотрены варианты составных спецификаторов.

(and *type**)

Соответствует пересечению типов *type*.

(array *type dimensions*)
(simple-array *type dimensions*)

Соответствуют множеству массивов с типом *type* и размерностями, соответствующими *dimensions*. Если *dimensions* – неотрицательное целое число, оно указывает на количество размерностей; если это список, то он указывает на величину каждой размерности (как в вызове `make-array`). Использование `simple-array` ограничивает множество простыми массивами. Знак `*`, встречающийся на месте типа *type* или одной из размерностей *dimensions*, означает отсутствие соответствующих ограничений.

(base-string *i*)
(simple-base-string *i*)

Эквиваленты (vector base-character *i*) и (simple-array base-character (*i*)) соответственно.

(bit-vector *i*)
(simple-bit-vector *i*)

Эквиваленты (array bit (*i*)) и (simple-array bit (*i*)) соответственно.

(complex *type*)

Соответствует множеству комплексных чисел, мнимые и действительные части которых принадлежат типу *type*.

(cons *type1 type2*)

Соответствует множеству ячеек, `car` которых принадлежит типу *type1*, а `cdr` – типу *type2*. Знак `*` в одной из позиций соответствует `t`.

(eql *object*)

Соответствует множеству из одного элемента: *object*.

(float *min max*)
(short-float *min max*)
(single-float *min max*)
(double-float *min max*)
(long-float *min max*)

Указывают на множество чисел с плавающей запятой определенного типа со значениями между *min* и *max*, где *min* и *max* – либо *f* (включая число *f*), либо (*f*) (исключая число *f*), где *f* – число соответствующего типа либо *, что указывает на отсутствие ограничения.

(function *parameters type*)

Используется только в декларациях. Соответствует множеству функций, чьи аргументы принадлежат типам, указанным *parameters*, и возвращающих значение(я) типа *type*. Список *parameters* соответствует списку спецификаторов типов для аргументов функции (аргументы по ключу сообщаются как список вида (*key type*)). Спецификатор, следующий за &rest, соответствует типу последующих аргументов, а не типу самого аргумента, который всегда является списком. (Также обратите внимание на спецификатор типа values.)

(integer *min max*)

Аналог float для целых чисел.

(member *object**)

Соответствует множеству объектов *object*.

(mod *i*)

Соответствует множеству целых чисел, меньших *i*.

(not *type*)

Соответствует дополнению до множества *type*.

(or *type**)

Соответствует объединению типов *type*.

(rational *min max*)

Аналог float для рациональных чисел.

(real *min max*)

Аналог float для действительных чисел.

(satisfies *symbol*)

Соответствует множеству объектов, удовлетворяющих функции с одним аргументом и именем *symbol*.

(signed-byte *i*)

Соответствует множеству целых чисел между $-2^{(i-1)}$ и $2^{(i-1)}-1$ включительно. Эквивалент integer, если *i* является *.

(string *i*)
(simple-string *i*)

Соответствуют множеству строк и простых строк длиной *i* соответственно.

(unsigned-byte *i*)

Соответствует множеству неотрицательных целых чисел, меньших 2^i. Эквивалент (integer 0 *), если *i* – *.

(values . *parameters*)

Используется только в спецификаторах function и выражениях the. Соответствует множеству наборов значений, которые могут передаваться в multiple-value-call с функцией типа (function *parameters*).

(vector *type i*)
(simple-vector *i*)

Эквиваленты (array *type* (*i*)) и (simple-array t (*i*)) соответственно. Необходимо помнить, что простой вектор – это не только простой одномерный массив. Простой вектор может также хранить объекты любых типов.

Макросы чтения

Макросами, состоящими из одного знака, являются (,), ', ; и `. Все предопределенные управляемые макросы чтения имеют управляющий знак #.

#*c*	Соответствует знаку *c*.
#'*f*	Эквивалент (function *f*).
#(...)	Соответствует простому вектору.
#n(...)	Соответствует простому вектору из *n* элементов. Если задано меньшее число элементов, недостающие заполняются значением последнего.
#*bbb*	Соответствует простому бит-вектору.
#*n**bbb*	Соответствует простому бит-вектору из *n* элементов. Если задано меньшее число элементов, недостающие заполняются значением последнего.
#:*sym*	Создает новый неинтернированный символ с именем *sym*.
#.*expr*	Создает значение *expr* на момент считывания.
#B*ddd*	Двоичное число.
#O*ddd*	Восьмеричное число.
#X*ddd*	Шестнадцатеричное число.

#*n*R*ddd*	Число с основанием *n*, которое должно быть десятичным целым числом от 2 до 36 включительно.
#C(*a b*)	Соответствует комплексному числу *a*+*bi*.
#*n*A*expr*	Соответствует многомерному массиву, созданному со значением '*expr* параметра :initial-contents в вызове make-array.
#S(*sym* ...)	Создает структуру типа *sym*, заполняя аргументы в соответствии с заданными значениями и присваивая остальным аргументам значения так же, как с помощью соответствующего конструктора.
#P*expr*	Эквивалент (parse-namestring '*expr*).
#*n*=*expr*	Эквивалент *expr*, но создается циклическая структура со ссылкой на элемент с меткой *n*.
#*n*#	Создает объект, являющийся меткой *n* в циклической структуре.
#+*test expr*	Если *test* пройден, то эквивалентен *expr*, иначе является пробелом.
#-*test expr*	Если *test* не пройден, то эквивалентен *expr*, иначе является пробелом.
#|...|#	Многострочный комментарий. Игнорируется считывателем.
#<	Вызывает ошибку.

Обратную кавычку легко понять, если рассмотреть значения, возвращаемые выражением, в котором она используется.° Чтобы вычислить такое выражение, удалим обратную кавычку и все соответствующие ей запятые, заменяя выражение после каждой кавычки на соответствующее ему значение. Вычисление выражения, начинающегося с запятой (вне обратной кавычки) приводит к ошибке.

Запятая соответствует обратной кавычке, когда между ними имеется одинаковое количество запятых и обратных кавычек (*b* находится между *a* и *c*, если *a* предшествует выражению, содержащему *b*, и *b* предшествует выражению, содержащему *c*). Это означает, что в корректном (well-formed) выражении последняя обратная кавычка совпадает с наиболее глубоколежащей запятой.

Предположим, что x вычисляется в a, которая вычисляется в 1; и y вычисляется в b, которая вычисляется в 2. Чтобы вычислить выражение:

```
``(w ,x ,,y)
```

удалим первую обратную кавычку и вычислим все, что следует за соответствующими ей запятыми. Ей соответствует лишь крайняя справа запятая. Если мы удалим ее и заменим оставшееся выражение его значением, то получим:

```
`(w ,x ,b)
```

Легко увидеть, что это выражение раскрывается в:

```
(w a 2)
```

Знак запятая-эт (`,@`) ведет себя аналогично запятой, но должен находиться в списке, а соответствующее выражение-аргумент должно вычисляться в список. Получаемый список вставляется поэлементно в список, содержащий выражение. Поэтому

```
``(w ,x ,,@(list 'a 'b))
```

вычисляется в

```
`(w ,x ,a ,b)
```

Знак запятая-точка (`,.`) аналогичен запятой-эт, но действует деструктивно.

Комментарии

Этот раздел может считаться библиографией. Все книги и статьи, перечисленные в нем, рекомендуются к прочтению.

14 Steele, Guy L., Jr., with Scott E. Fahlman, Richard P. Gabriel, David A. Moon, Daniel L. Weinreb, Daniel G. Bobrow, Linda G. DeMichiel, Sonya E. Keene, Gregor Kiczales, Crispin Perdue, Kent M. Pitman, Richard C. Waters, and Jon L White. «Common Lisp: the Language», 2nd Edition. Digital Press, Bedford (MA), 1990.

19 McCarthy, John. «Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I». *CACM*, 3:4 (April 1960), pp. 184–195.

McCarthy, John. «History of Lisp». In Wexelblat, Richard L. (Ed.) «History of Programming Languages». Academic Press, New York, 1981, pp. 173–197.

Эти книги доступны по адресу *http://www-formal.stanford.edu/jmc/*.

21 Brooks, Frederick P. «The Mythical Man-Month»[1]. Addison-Wesley, Reading (MA), 1975, p. 16.

Быстрое прототипирование – это не просто способ написания качественных программ в краткие сроки. Это способ написания программ, которые без него не будут написаны вовсе.

Даже очень амбициозные люди начинают с небольших дел. Проще всего сначала заняться несложной (или, по крайней мере, кажущейся несложной) задачей, которую вы в состоянии сделать в одиночку. Вот почему многие большие дела происходили из малых начинаний. Быстрое прототипирование дает возможность начать с малого.

22 Там же, p. i.

23 Murray, Peter and Linda. «The Art of the Renaissance». Thames and Hudson, London, 1963, p. 85.

23 Janson, W J. «History of Art», 3rd Edition. Abrams, New York, 1986, p. 374.

Аналогия применима, разумеется, только к станковой живописи и позднее к живописи по холсту. Настенная живопись по-прежнему выполнялась в виде фресок. Я также не берусь утверждать, что стили живописи полностью следовали за технологическим прогрессом, наоборот, противоположное мнение кажется более отвечающим действительности.

30 Имена `car` и `cdr` происходят из внутреннего представления списков в первой реализации Лиспа: `car` соответствовал «Contents of Address part of Register»,

---

[1] Фредерик Брукс «Мифический человеко-месяц, или как создаются программные системы». – Пер. с англ. – СПб.: Символ-Плюс, 2000.

то есть содержимому адресной части регистра, а `cdr` – «Contents of the Decrement part of the Register», то есть декрементной части регистра.

34 Читатели, столкнувшиеся с серьезными затруднениями в понимании рекурсии, могут обратиться к одной из следующих книг:

Touretzky, David S. «Common Lisp: A Gentle Introduction to Symbolic Computation». Benjamin/Cummings, Redwood City (CA), 1990, Chapter 8.

Friedman, Daniel P., and Matthias Felleisen. «The Little Lisper». MIT Press, Cambridge, 1987.

43 В ANSI Common Lisp имеется также макрос `lambda`, позволяющий писать `(lambda (x) x)` вместо `#'(lambda (x) x)`. Поскольку он скрывает симметрию между лямбда-выражениями и символьными именами функций (для которых необходимо использовать `#'`), его использование привносит довольно сомнительную элегантность.

44 Gabriel, Richard P. «Lisp: Good News, Bad News, How to Win Big». *AI Expert*, June 1991, p. 34.

62 Следует остерегаться еще одного момента при использовании `sort`: она не гарантирует сохранение порядка следования элементов, равных с точки зрения функции сравнения. Например, при сортировке `(2 1 1.0)` с помощью `<` корректная реализация Common Lisp может вернуть как `(1 1.0 2)`, так и `(1.0 1 2)`. Чтобы сохранить исходный порядок для одинаковых с точки зрения функции сравнения элементов, используйте более медленную функцию `stable-sort` (также деструктивную), которая возвращает только первое значение.

76 Много было сказано о пользе комментариев и практически ничего – о затратах на них. Но комментарии имеют свою цену. Хороший код, как хорошая проза, требует постоянного пересмотра. Для этого код должен быть компактным и податливым. Межстрочные комментарии делают код менее гибким и более расплывчатым, задерживая тем самым его развитие.

77 Несмотря на то, что подавляющее число реализаций используют набор знаков ASCII, Common Lisp гарантирует лишь следующее их упорядочение: 26 знаков нижнего регистра, расположенных по возрастанию в алфавитном порядке, их аналоги в верхнем регистре, а также цифры от 0 до 9.

90 Стандартный способ реализации очередей с приоритетом – использование структуры, называемой *кучей*. См. Sedgewick, Robert. «Algorithms». Addison-Wesley, Reading (MA), 1988.

95 Определение `progn` похоже на правило вычисления вызовов функций в Common Lisp (стр. 27). Хотя `progn` и является специальным оператором, мы можем определить похожую функцию:

```
(defun our-progn (&rest args)
  (car (last args)))
```

Она будет совершенно неэффективна, но функционально эквивалентна реальному `progn`, если последний аргумент возвращает только одно значение.

98 Аналогия с лямбда-выражениями нарушается, если именами переменных являются символы, имеющие в списке параметров особое значение. Например, выражение:

```
(let ((&key 1) (&optional 2)))
```

корректно, а вот соответствующее ему лямбда-выражение – уже нет:

```
((lambda (&key &optional)) 1 2)
```

При построении аналогии между `do` и `labels` возникает такая же проблема. Выражаю благодарность Дэвиду Кужнику (David Kuzhnik) за указание на нее.

102 Steele, Guy L., Jr., and Richard P. Gabriel. «The Evolution of Lisp». *ACM SIGPLAN Notices* 28:3 (March 1993). Пример, процитированный в этом отрывке, перенесен в Common Lisp из Scheme.

104 Чтобы сделать вывод времени более читаемым, необходимо убедиться, что минуты и секунды представляются в виде пары цифр:

```
(defun get-time-string ()
  (multiple-value-bind (s m h) (get-decoded-time)
    (format nil "~A:~2,,,'0@A:~2,,,'0@A" h m s)))
```

107 В своем письме от 18 марта (по старому стилю) 1751 года Честерфилд писал:

> «Хорошо известно, что Юлианский календарь неточен, так как переоценивает продолжительность года на одиннадцать дней. Папа Григорий Тринадцатый исправил эту ошибку [в 1582]; его календарная реформа немедленно была принята в Католической Европе, а затем и протестантами, но не была принята в России, Швеции и Англии. На мой взгляд, отрицание грубой общеизвестной ошибки не делает Англии особой чести, особенно в такой компании. Вытекающие из этого факта неудобства испытывали, например, те, кто имел дело с иностранной корреспонденцией, политической или финансовой. По этой причине я предпринял попытку реформирования: проконсультировался с лучшими юристами и опытнейшими астрономами, и мы подготовили законопроект по этому вопросу. Но здесь начались первые трудности: я должен был использовать юридический язык и астрономические вычисления, но в этих вопросах я дилетант. Тем не менее было совершенно необходимо, чтобы в Палате Лордов меня сочли разбирающимся в этих вопросах и чтобы они сами не чувствовали себя профанами. Со своей стороны, я уже изъяснялся с ними на Кельтском и Славонском на темы, связанные с астрономией, и они вполне поняли меня. Поэтому я решил, что лучше заниматься делом, а не разглагольствовать и угождать, вместо того чтобы информировать. Я лишь дал им небольшой экскурс в историю календарей, от Египетского к Грегорианскому, развлекая их занимательными историческими эпизодами. При этом я был крайне осмотрителен в выборе слов и применил свое ораторское искусство, чтобы представить гармоничность и завершенность интересующего меня случая. Это сработало, и сработало бы всегда. Я усладил их слух и донес нужную информацию, после чего многие из них сообщили, что проблема теперь совершенно ясна. Но, видит Бог, я даже не пытался ничего доказывать.»

См. Roberts, David (Ed.) «Lord Chesterfield's Letters». Oxford University Press, Oxford, 1992.

108 В Common Lisp универсальным временем считается целое число, представляющее количество секунд, прошедших с начала 1900 года. Функции `encode-universal-time` и `decode-universal-time` осуществляют перевод дат между форматами. Таким образом, для дат после 1900 года в Common Lisp есть более простой способ выполнения этого преобразования:

```
(defun num->date (n)
  (multiple-value-bind (ig no re d m y)
                       (decode-universal-time n)
    (values d m y)))

(defun date->num (d m y)
  (encode-universal-time 1 0 0 d m y))

(defun date+ (d m y n)
  (num->date (+ (date->num d m y)
                (* 60 60 24 n))))
```

Помимо вышеупомянутого ограничения имеется предел, за которым представления дат не будут принадлежать типу `fixnum`.

112 Хотя вызов `setf` может пониматься как обращение к ссылке на конкретное место, он реализует более обобщенный механизм. Пусть, например, `marble` – это структура с одним полем `color`:

```
(defstruct marble
  color)
```

Следующая функция принимает список разновидностей мрамора (marble) и возвращает их цвет, если он один для всех экземпляров, и `nil` – в противном случае:

```
(defun uniform-color (lst)
  (let ((c (marble-color (car lst))))
    (dolist (m (cdr lst))
      (unless (eql (marble-color m) c)
        (return nil)))
    c))
```

Хотя `uniform-color` не ссылается на конкретное место, возможно и разумно вызывать `setf` с этой функцией в качестве первого аргумента. Определив

```
(defun (setf uniform-color) (val lst)
  (dolist (m lst)
    (setf (marble-color m) val)))
```

мы сможем сказать

```
(setf (uniform-color *marbles*) 'red)
```

чтобы сделать каждый элемент в `*marbles*` красным.

112 В ранних реализациях Common Lisp приходилось использовать `defsetf` для определения раскрытия `setf`-выражения. При определении порядка следования аргументов необходимо проявлять аккуратность и помнить, что новое значение располагается *последним* в определении функции, передаваемой вторым аргументом `defsetf`. Таким образом, вызов:

```
(defun (setf primo) (val lst) (setf (car lst) val))
```

эквивалентен

```
(defsetf primo set-primo)

(defun set-primo (lst val) (setf (car lst) val))
```

118 Язык C, например, позволяет передавать указатель на функцию, но при этом список того, что вы можете передать функции, существенно меньше (т. к. в C нет замыканий), как и список действий, которые может совершить с этой функцией получатель (т. к. в C нет аналога `apply`). Кроме того, вы обязаны декларировать для нее тип возвращаемого значения. Сможете ли вы написать в C аналог `map-int` или `filter`? Разумеется, нет. Вам придется подавлять проверку типов аргументов и возвращаемого значения, а это опасно и, вероятно, возможно лишь для 32-битных значений.

120 За разнообразными примерами всестороннего использования замыканий обращайтесь к книге Abelson, Harold, and Gerald Jay Sussman, with Julie Sussman. «Structure and Interpretation of Computer Programs».[1] MIT Press, Cambridge, 1985.

120 За дополнительной информацией о языке Dylan обращайтесь к книге Shalit, Andrew, with Kim Barrett, David Moon, Orca Starbuck, and Steve Strassmann. «Dylan Interim Reference Manual». Apple Computer, 1994. Этот документ доступен[2] на нескольких ресурсах, в том числе *http://www.harlequin.com* и *http://www.apple.com*.

Scheme – это очень маленький и чистый диалект Лиспа. Он был придуман Гаем Стилом и Джеральдом Сассманом в 1975 году и на момент написания книги был определен в книге Clinger, William, and Jonathan A. Rees (Eds.) «Revised Report on the Algorithmic Language Scheme»[3]. 1991.

Этот отчет, а также разнообразные реализации Scheme на момент печати были доступны через анонимный FTP: *swiss-ftp.ai.mit.edu:pub*.

Можно отметить две замечательные книги по языку Scheme: «Structure and Interpretation of Computer Programs» («Структура и интерпретация компьютерных программ»), указанная ранее, и Springer, George and Daniel P. Friedman «Scheme and the Art of Programming». MIT Press, Cambridge, 1989.

123 Наиболее неприятные баги в Лиспе могут быть связаны с динамическим диапазоном. Подобные вещи практически никогда не возникнут в Common Lisp, который по умолчанию использует лексический диапазон. Многие диалекты Лиспа, используемые, как правило, как встраиваемые языки, работают с динамическим диапазоном, и об этом необходимо помнить.

Один подобный баг напоминает проблему захвата переменных (стр. 176). Представьте, что вы передаете одну функцию как аргумент другой, и эта функция получает некоторую переменную в качестве аргумента. Однако внутри функ-

---

[1] Эта книга очень полезна. В Интернете имеется русский перевод (под названием «Структура и интерпретация компьютерных программ»; книга также известна как SICP), решения большинства задач из книги, а также соответствующие видеолекции MIT с русскими субтитрами. – *Прим. перев.*

[2] На момент подготовки перевода ни один из этих ресурсов, похоже, не содержал упомянутого материала. Документация и прочая актуальная информация о языке Dylan доступна по адресу *http://www.opendylan.org*. – *Прим. перев.*

[3] На момент подготовки перевода актуальной являлась шестая версия спецификации (R6RS), доступная по адресу *http://www.r6rs.org*. – *Прим. перев.*

ции, которая ее вызывает, данная переменная не определена или имеет другое значение.

Предположим, к примеру, что мы написали ограниченную версию mapcar:

```
(defun our-mapcar (fn x)
  (if (null x)
      nil
      (cons (funcall fn (car x))
            (our-mapcar fn (cdr x)))))
```

Пусть эта функция используется в другой, add-to-all, которая принимает число и добавляет его к каждому элементу списка:

```
(defun add-to-all (lst x)
  (our-mapcar #'(lambda (num) (+ num x))
              lst))
```

В Common Lisp этот код будет работать без проблем, но в Лиспе, использующем динамический диапазон, мы получили бы ошибку. Функция, передаваемая как аргумент our-mapcar, ссылается на x. В момент передачи этой функции в our-mapcar переменная x должна быть числом, переданным в качестве второго аргумента add-to-all. Но внутри вызова функции our-mapcar переменная x будет иметь иное значение, а именно список, передаваемый как второй аргумент our-mapcar. Когда этот список будет передан вторым аргументом +, будет вызвана ошибка.

134 Более новые реализации Common Lisp содержат переменную *read-eval*, которая может отключать макрос чтения #.. При вызове read-from-string с пользовательским вводом будет разумно связать *read-eval* с nil. В противном случае пользователь может получить побочные эффекты, если будет использовать #. во вводе.

136 Существует множество оригинальных алгоритмов быстрого поиска совпадений в строках, но применительно к поиску в текстовых файлах наш метод грубой силы оказывается вполне быстрым. За другими алгоритмами поиска совпадений в строках обращайтесь к книге Sedgewick, Robert «Algorithms». Addison-Wesley, Reading (MA), 1988.

151 В 1984 году Common Lisp определял reduce без параметра по ключу :key, поэтому random-next нужно было бы определить следующим образом:

```
(defun random-next (prev)
  (let* ((choices (gethash prev *words*))
         (i (random (let ((x 0))
                      (dolist (c choices)
                        (incf x (cdr c)))
                      x))))
    (dolist (pair choices)
      (if (minusp (decf i (cdr pair)))
          (return (car pair))))))
```

151 Программа, подобная Henley, использовалась в 1989 году для симуляции новостных лент в интернет-рассылках авторства известных флеймеров. Значительное количество читателей приняло подобные сообщения за правду. Как и за всеми хорошими розыгрышами, за этим стояла определенная идея. Что можно

сказать о содержании реальных флеймов или о внимании, с которым их читают, если случайно сгенерированный текст мог быть принят за реальное сообщение?

Серьезным вкладом в развитие искусственного интеллекта можно считать умение разделять задачи по сложности. Некоторые задачи оказываются вполне тривиальными, другие – почти неразрешимыми. Поскольку работы, связанные с искусственным интеллектом, посвящены в первую очередь последним, изучение первых можно было бы назвать *искусственной глупостью* (*artificial stupidity*). Возможно, глупое имя, но у этой области есть большие перспективы – она может дать нам программы, подобные Henley, которые будут выполнять роль контрольных задач для проверки каких-либо гипотез.

Оказывается, что иногда слова вовсе не обязаны передавать какие-либо мысли, и этому способствует склонность людей усиленно искать смысл там, где его нет. Поэтому если оратор позаботится о привнесении малейшего смысла в свою речь, доверчивая публика несомненно сочтет их *глубокомысленными*.

Пожалуй, это явление ничуть не моложе самого человечества. Однако новые технологии позволяют шагнуть еще дальше, и примером этого может служить наша программа генерации случайного текста. Она предельно проста, но может заставить людей искренне считать свою «поэзию» делом рук человеческих (можете проверить на своих друзьях). Более сложные программы, несомненно, смогут генерировать более осмысленные тексты.[1]

Интересное обсуждение случайной генерации стихов как полноценной литературной формы вы можете найти здесь: Low, Jackson M. Poetry, Chance, Silence, Etc. In Hall, Donald (Ed.) «Claims for Poetry». University of Michigan Press, Ann Arbor, 1982.

Стараниями Online Book Initiative текстовые версии многих классических работ, включая эту, доступны в сети. На момент издания оригинала[2] они были доступны по анонимному FTP: *ftp.std.com:obi*.

Также вам может показаться интересным режим Dissociated Press в Emacs.

161 Следующая функция отображает значения 16 констант, определяющих ограничения на размер чисел с плавающей запятой в используемой реализации:

```
(defun float-limits ()
  (dolist (m '(most least))
    (dolist (s '(positive negative))
```

[1] Описанная идея породила немало других шуток, в том числе в России. Многие из них играют на псевдонаучности получаемого текста. Например, сервис Яндекс. Рефераты предлагает генерацию рефератов на заданные темы. Скандалом обернулся также эксперимент М. Гельфанда, в котором псевдонаучная статья под названием «Корчеватель: алгоритм типичной унификации точек доступа и избыточности», сгенерированная программой SCIgen (написанной в MIT) и переложенная на русский язык с помощью отечественной разработки «Этап-3», была допущена к печати в одном из журналов ВАК (в 2008 г). Подобная шутка была позже организована в одном из американских журналов. – *Прим. перев.*

[2] Похоже, этот документ, как и сама организация OBI, канул в лету. – *Прим. перев.*

```
(dolist (f '(short single double long))
  (let ((n (intern (string-upcase
                    (format nil "~A-~A-~A-float"
                                m  s  f)))))
    (format t "~30A ~A~%" n (symbol-value n)))))))
```

174 Алгоритм quicksort был предложен Хоаром в 1962 году и описан в книге Дональда Кнута «Sorting and Searching». Addison-Wesley, Reading (MA), 1973.

182 Foderaro, John K. «Introduction to the Special Lisp Section». *CACM* 34:9 (September 1991), p. 27.

185 За детальной информацией об использовании CLOS обращайтесь к книгам:

Keene, Sonya E. «Object Oriented Programming in Common Lisp». Addison-Wesley, Reading (MA), 1989.

Kiczales, Gregor, Jim des Rivieres, and Daniel G. Bobrow. «The Art of the Metaobject Protocol». MIT Press, Cambridge, 1991.

187 Давайте скажем на другой лад: *мы можем выполнить модификации такого рода даже без просмотра остального кода*. Эти слова могут прозвучать тревожно для некоторых читателей. В них заключается рецепт спагетти-кода.

Объектно-ориентированная модель позволяет упростить построение программ с помощью их расширения. Такая возможность в неумелых руках может привести к спагетти-коду. Это не всегда плохо, но ничего хорошего в таком явлении тоже нет.

Приличная часть когда-либо написанного кода является именно спагетти-кодом, и положение дел вряд ли поменяется в скором времени. Для программ, которые в любом случае закончат как спагетти, объектно-ориентированная модель, несомненно, полезна: они хотя бы будут структурированными спагетти. Но для программ, которые желали бы избежать подобной участи, использование объектной ориентированности может быть скорее опасным, чем полезным.

192 При наследовании экземпляром слота с одинаковым именем от нескольких суперклассов наследуемый слот содержит суперпозицию свойств всех родительских слотов. Механизм комбинации может различаться для разных свойств:

1. Свойства :allocation, :initform (если имеется) и :documentation (если имеется) будут получены из наиболее специфичного класса.
2. Свойства :initarg будут объединением параметров :initarg всех суперклассов. Это справедливо также для свойств :accessor, :reader и :writer.
3. Свойство :type будет пересечением параметров :type всех суперклассов.

199 Вы можете избежать необходимости удалять символы вручную, изначально используя неинтернированные символы:

```
(progn
  (defclass counter () ((#1=#:state :initform 0)))

  (defmethod increment ((c counter))
    (incf (slot-value c '#1#)))

  (defmethod clear ((c counter))
    (setf (slot-value c '#1#) 0)))
```

Особого смысла progn здесь не несет, он просто показывает, что эти выражения должны выполняться вместе в заданном порядке. Если для вас это неудобно, можете воспользоваться следующим макросом чтения:

```
(defvar *symtab* (make-hash-table :test #'equal))

(defun pseudo-intern (name)
  (or (gethash name *symtab*)
      (setf (gethash name *symtab*) (gensym))))

(set-dispatch-macro-character #\# #\[
  #'(lambda (stream charl char2)
      (do ((acc nil (cons char acc))
           (char (read-char stream) (read-char stream)))
          ((eql char #\]) (pseudo-intern acc)))))
```

Теперь вы сможете сделать следующее:

```
(defclass counter () ((#[state] :initform 0)))

(defmethod increment ((c counter))
  (incf (slot-value c '#[state])))

(defmethod clear ((c counter))
  (setf (slot-value c '#[state]) 0))
```

210 Данный макрос добавляет новый элемент в двоичное дерево поиска:

```
(defmacro bst-push (obj bst <)
  (multiple-value-bind (vars forms var set access)
                       (get-setf-expansion bst)
    (let ((g (gensym)))
      `(let* ((,g ,obj)
              ,@(mapcar #'list vars forms)
              (,(car var) (bst-insert! ,g ,access ,<)))
         ,set))))
```

220 Knuth, Donald E. «Structured Programming with goto Statements». *Computing Surveys*, 6:4 (December 1974), pp. 261–301.

221 Knuth, Donald E. «Computer Programming as an Art». *In ACM Turing Award Lectures: The First Twenty Years*. ACM Press, 1987.

Эта статья, как и предыдущая, была переиздана в: Knuth, Donald E. «Literate Programming». CSLI Lecture Notes #27, Stanford University Center for the Study of Language and Information, Palo Alto, 1992.

223 Steele, Guy L., Jr. «Debunking the "Expensive Procedure Call" Myth or, Procedural Call Implementations Considered Harmful or, LAMBDA: The Ultimate GOTO». *Proceedings of the National Conference of the ACM*, 1977, p. 157.

Суть оптимизации хвостовой рекурсии заключается в замене рекурсивных вызовов аналогичным итеративным выражением. К сожалению, многие компиляторы, актуальные на момент написания книги, генерировали для циклов все же более быстрый код.

224 Некоторые примеры использования disassemble на различных процессорах вы сможете найти в книге Norvig, Peter «Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp». Morgan Kaufmann, San Mateo (CA), 1992.

225 Популярность объектно-ориентированного программирования часто связывают с популярностью C++, которая возникла благодаря усовершенствованиям в плане типизации. Концептуально C++ связан с C, но в числе прочего позволяет создавать операторы, работающие с аргументами разных типов. Однако для этого вовсе не обязательно использовать объектно-ориентированный язык – достаточно динамической типизации. Если вы понаблюдаете за людьми, пишущими на C++, то обнаружите, что создаваемые ими иерархии классов обычно являются плоскими. C++ стал популярным не потому, что позволяет работать с классами и методами, а потому что людям необходимо обходить некоторые ограничения типизации в C.

226 Создание деклараций значительно упрощается с помощью макросов. Следующий макрос получает имя типа и выражение (вероятно, численное) и генерирует раскрытие, в котором задекларирован заданный тип для всех промежуточных результатов. Чтобы гарантировать вычисление выражения *e* с помощью только fixnum-арифметики, достаточно сказать (with-type fixnum *e*).

```
(defmacro with-type (type expr)
  `(the ,type ,(if (atom expr)
                   expr
                   (expand-call type (binarize expr)))))

(defun expand-call (type expr)
  `(,(car expr) ,@(mapcar #'(lambda (a)
                              `(with-type ,type ,a))
                          (cdr expr))))

(defun binarize (expr)
  (if (and (nthcdr 3 expr)
           (member (car expr) '(+ - * /)))
      (destructing-bind (op a1 a2 . rest) expr
        (binarize `(,op (,op ,a1 ,a2) ,@rest)))
      expr))
```

Вызов binarize гарантирует, что никакой арифметический оператор не вызывается более чем с двумя аргументами. Вызов типа:

```
(the fixnum (+ (the fixnum a)
               (the fixnum b)
               (the fixnum c)))
```

не может быть скомпилирован в сложения чисел типа fixnum, так как промежуточный результат (например, a+b) может не принадлежать типу fixnum.

С помощью with-type мы можем заменить полную деклараций версию poly (стр. 226) на более простую:

```
(defun poly (a b x)
  (with-type fixnum (+ (* a (expt x 2)) (* b x))))
```

Если вам предстоит много работать с fixnum-арифметикой, возможно, стоит определить макрос чтения, раскрывающийся в вызов (`with-type fixnum ...`).

231 На многих Unix-системах пригодным файлом со словами является `/usr/dict/words`.

233 T является диалектом Scheme с множеством полезных дополнений, включая поддержку пулов. Больше информации по этой теме вы можете найти в книге Rees, Jonathan A., Norman I. Adams, and James R.Meehan. «The T Manual», 5th Edition. Yale University Computer Science Department, New Haven, 1988.

На момент публикации оригинала[1] руководство по T, а также его реализация были доступны по анонимному FTP: *hing.lcs.mit.edu:pub/t3.1*.

237 Различия между спецификациями и программами можно считать количественными, но не качественными. Однажды поняв это, становится неестественным *требовать*, чтобы кто-либо составлял документацию до начала реализации проекта. Если программа пишется на низкоуровневом языке, то будет полезно предварительно описать задачу в терминах высокого уровня. Но при переходе к высокоуровневым языкам необходимость в подобных действиях исчезает. В некотором роде реализация и промежуточная спецификация становятся одним и тем же.

Если программа, ранее написанная на низкоуровневом языке, переписывается на более абстрактном языке, ее реализация начинает напоминать спецификацию даже в большей степени, чем использовавшаяся ранее спецификация. Перефразируя основную мысль раздела 13.7, можно сказать: спецификация программы на C может быть написана на Лиспе.

237 История отливки скульптуры «Персей» Бенвенутто Челлини, вероятно, наиболее известная (и наиболее курьезная), по сравнению с другими литыми бронзовыми скульптурами. Cellini, Benvenuto. «Autobiography». Перевод George Bull, Penguin Books, Harmondsworth, 1956.

246 Даже опытные Лисп-программисты иногда находят работу с пакетами неудобной. Потому ли это, что они и правда сложны, или же мы просто не привыкли думать, что происходит в момент чтения?

Подобная концептуальная трудность также связана с `defmacro`. Большой объем работы был выполнен для поиска более абстрактной альтернативы `defmacro`. Однако `defmacro` кажется сложным, только если использовать его, придерживаясь довольно распространенного представления о том, что определение макроса подобно определению функции. В этом случае может шокировать, что приходится думать о проблеме захвата переменных. Но если размышлять о макросах как о том, чем они действительно являются, т. е. как о преобразованиях исходного кода, то захват переменных будет не большей проблемой, чем деление на ноль.

Так что, возможно, пакеты окажутся разумным способом реализации модульности. Даже при беглом взгляде очевидно, что они сильно напоминают методи-

[1] На момент подготовки перевода этот ресурс не был доступен. Данный документ легко отыскать в сети, но не похоже, чтобы этот диалект представлял на настоящий момент какой-либо существенный интерес. – *Прим. перев.*

ки, используемые программистами при отсутствии формальной модульной системы.[1]

248 Можно утверждать, что макрос `loop` является более общим и что не стоит определять множество операторов, чтобы добиться того, что можно получить и с одним. Но `loop` является одним оператором только с формальной точки зрения. В этом смысле `eval` также является одним оператором. Но с точки зрения концептуальной сложности для пользователя `loop` содержит, по меньшей мере, столько же операторов, сколько и различных типов предложений. К тому же эти предложения не могут использоваться отдельно как полноценные операторы в Лиспе: невозможно взять часть `loop`-выражения и передать его как аргумент какой-то функции, как это можно сделать с `map-int`, например.

254 Больше информации по теме логического вывода можно найти в книге Russell, Stuart, and Peter Norvig «Artificial Intelligence: A Modern Approach»[2]. Prentice Hall, Englewood Cliffs (NJ), 1995.

278 Программа в главе 17 использует возможность формы `setf` быть первым аргументом `defun`. Она отсутствовала в ранних реализациях Common Lisp, поэтому для запуска приведенного кода на ранних реализациях потребуется небольшая модификация:

```
(proclaim '(inline lookup set-lookup))

(defsetf lookup set-lookup)

(defun set-lookup (prop obj val)
  (let ((off (position prop (layout obj) :test #'eq)))
    (if off
        (setf (svref obj (+ off 3)) val)
        (error "Can't set "A of ~A." val obj))))

(defmacro defprop (name &optional meth?)
  `(progn
     (defun ,name (obj &rest args)
       ,(if meth?
            `(run-methods obj ',name args)
            `(rget ',name obj nil)))
  (defsetf ,name (obj) (val)
    `(setf (lookup ',',name ,obj) ,val))))
```

---

[1] Это «пророчество» оказалось верным, и на сегодняшний момент пакеты используются повсеместно и полноценно в Common Lisp-коде и признаются одной из важнейших особенностей языка. – *Прим. перев.*

[2] Книга переведена на русский: Рассел С., Норвиг П. «Искусственный интеллект: современный подход», 2-е изд. – Пер. с англ. и ред. К.А. Птицына. – М.: Вильямс, 2006.
Книга (известная также как AIMA) представляет собой фундаментальный труд, посвященный проблемам ИИ. Она ценна тем, что дает систематизированное представление об ИИ и объединяет основные его концепции, но вряд ли может служить полноценным руководством по какой-либо отдельно взятой области ИИ. Вас также может заинтересовать код на Common Lisp, сопровождающий книгу. Он свободно доступен по адресу: *http://aima.cs.berkeley.edu/lisp/*. – *Прим. перев.*

281 Если бы defmeth была определена следующим образом:

```
(defmacro defmeth (name obj parms &rest body)
  (let ((gobj (gensym)))
    `(let ((,gobj ,obj))
       (setf (gethash ',name ,gobj)
             #'(lambda ,parms
                 (labels ((next ()
                            (funcall (get-next ,gobj ',name)
                                     ,@parms)))
                   ,@body))))))
```

то вызывать следующий метод можно было бы просто через next:

```
(defmeth area grumpy-circle (c)
  (format t "How dare you stereotype me!~%")
  (next))
```

Выглядит проще. Однако, чтобы выполнять работу next-method-p, нам придется определить еще одну функцию.

288 Для по-настоящему быстрого доступа к слотам может пригодиться следующий макрос:

```
(defmacro with-slotref ((name prop class) &rest body)
  (let ((g (gensym)))
    `(let ((,g (+ 3 (position ,prop (layout ,class)
                              :test #'eq))))
       (macrolet ((,name (obj) `(svref ,obj ,',g)))
         ,@body))))
```

Он определяет локальный макрос, ссылающийся непосредственно на элемент вектора, соответствующий слоту. Попытка доступа к слоту будет оттранслирована непосредственно в вызов svref.

Пусть, например, определен класс balloon:

```
(setf balloon-class (class nil size))
```

Тогда мы сможем воспользоваться нашим макросом, чтобы быстро лопнуть все шарики (balloons).

```
(defun popem (balloons)
  (with-slotref (bsize 'size balloon-class)
    (dolist (b balloons)
      (setf (bsize b) 0))))
```

288 Gabriel, Richard P. «Lisp: Good News, Bad News, How to Win Big». *AI Expert*, June 1991, p. 35.

Еще в далеком 1973 году Ричард Фейтман сумел показать, что компилятор MacLisp для PDP-10 производил более быстрый код, чем компилятор Фортрана от производителя этого компьютера (т. е. DEC). См. Fateman, Richard J. «Reply to an editorial». *ACM SIGSAM Bulletin*, 25 (March 1973), pp. 9–11.

419 С помощью следующей идеи проще понять суть обратной кавычки. Будем рассматривать ее подобно обычной, а `,x будем раскрывать в (bq (comma x)). Тогда

мы сможем обработать обратную кавычку, расширяя eval, как показано в следующем наброске:

```
(defun eval2 (expr)
  (case (and (consp expr) (car expr))
    (comma (error "unmatched comma"))
    (bq    (eval-bq (second expr) 1))
    (t     (eval expr))))

(defun eval-bq (expr n)
  (cond ((atom expr)
         expr)
        ((eql (car expr) 'comma)
         (if (= n 1)
             (eval2 (second expr))
             (list 'comma (eval-bq (second expr)
                                   (1- n)))))
        ((eql (car expr) 'bq)
         (list 'bq (eval-bq (second expr) (1+ n))))
        (t
         (cons (eval-bq (car expr) n)
               (eval-bq (cdr expr) n)))))
```

Параметр n, используемый в eval-bq, применяется для нахождения совпадающих на текущий момент запятых. Каждая обратная кавычка увеличивает значение n, а каждая запятая – уменьшает. Запятая, встреченная при n = 1, считается совпадающей.

Вот пример со стр. 419:

```
> (setf x 'a a 1 y 'b b 2)
2
> (eval2 '(bq (bq (w (comma x) (comma (comma y))))))
(BQ (W (COMMA X) (COMMA B)))
> (eval2 *)
(W A 2)
```

В некоторый момент случайным образом образовалась особенная молекула. Назовем ее *Репликатором*. Она не обязательно была самой большой или самой сложной среди себе подобных, но она обладала особым свойством – умением копировать саму себя.

Ричард Докинз (Richard Dawkins)
«The Selfish Gene» (Эгоистичный Ген)

Для начала нам следует определить набор символьных выражений в терминах ориентированных пар и списков. Затем нужно определить пять элементарных функций и предикатов и построить на их основе условные выражения и рекурсивные определения более широкого набора функций (мы приведем ряд примеров с ними). Затем мы покажем, как сами эти функции могут быть выражены в символьном виде, а также определим универсальную функцию `apply`, позволяющую вычислять значение заданной функции с заданными аргументами.

Джон Маккарти (John McCarthy)
«Recursive Functions of Symbolic Expressions
and their Computation by Machine, part I»
(Рекурсивные функции в символьных выражениях
и их аппаратное вычисление, часть I)

Алфавитный указатель

Символы

*, переменная, 415
*, функция, 365
**, переменная, 415
***, переменная, 415
/, переменная, 416
/, функция, 365
//, переменная, 416
///, переменная, 416
+, переменная, 415
+, функция, 365
++, переменная, 415
+++, переменная, 415
–, переменная, 416
–, функция, 365
/=, функция, 364
<, функция, 365
<=, функция, 365
=, функция, 364
>, функция, 365
>=, функция, 365
#A, макрос чтения, 419
#B, макрос чтения, 418
#C, макрос чтения, 419
#n#, макрос чтения, 419
#n=, макрос чтения, 419
#O, макрос чтения, 418
#P, макрос чтения, 419
#R, макрос чтения, 419
#S, макрос чтения, 419
#X, макрос чтения, 418
;, макрос чтения, 418
‘, макрос чтения, 418
‘, макрос чтения, 418
(, макрос чтения, 418
), макрос чтения, 418
макрос чтения, 418
#, макрос чтения, 419
#:, макрос чтения, 418
#., макрос чтения, 418
#’, макрос чтения, 418
#(, макрос чтения, 418
#*, макрос чтения, 418
#+, макрос чтения, 419
#<, макрос чтения, 419
#|, макрос чтения, 419

Числа

1–, функция, 365
1+, функция, 365

A

abort, функция, 346
abs, функция, 356
acons, функция, 367
acosh, функция, 356
acos, функция, 356
add-method, обобщенная функция, 335
adjoin, функция, 368
adjustable-array-p, функция, 373
adjust-array, функция, 373
allocate-instance, обобщенная функция, 335
alpha-char-p, функция, 365
alphanumericp, функция, 365
and, макрос, 321
append, функция, 368
apply, функция, 321
apropos-list, функция, 406
apropos, функция, 406
aref, функция, 374
arithmetic-error-operands, функция, 356
arithmetic-error-operation, функция, 356
array-dimension, функция, 374
array-dimension-limit, константа, 409
array-dimensions, функция, 374
array-displacement, функция, 374
array-element-type, функция, 374
array-has-fill-pointer-p, функция, 374
array-in-bounds-p, функция, 375
arrayp, функция, 375
array-rank-limit, константа, 409

array-rank, функция, 375
array-row-major-index, функция, 375
array-total-size-limit, константа, 409
array-total-size, функция, 375
ash, функция, 357
asin, функция, 357
asinh, функция, 357
assert, макрос, 346
assoc, функция, 368
assoc-if, функция, 368
assoc-if-not, функция, 368
atan, функция, 357
atanh, функция, 357
atom, функция, 368
Autocad, 21

B

break-on-signals, переменная, 409
bit, функция, 375
bit-and, функция, 375
bit-andc1, функция, 375
bit-andc2, функция, 375
bit-eqv, функция, 375
bit-ior, функция, 375
bit-nand, функция, 375
bit-nor, функция, 375
bit-not, функция, 375
bit-orc1, функция, 376
bit-orc2, функция, 376
bit-vector-p, функция, 376
bit-xor, функция, 376
block, специальный оператор, 321
boole, функция, 357
boole-1, константа, 409
boole-2, константа, 409
boole-xor, константа, 409
both-case-p, функция, 365
boundp, функция, 351
break, функция, 347
break loop, 28
broadcast-stream-streams, функция, 389
butlast, функция, 368
byte, функция, 358
byte-position, функция, 358
byte-size, функция, 358

C

C, язык программирования
  выразительная сила, 19
  сложный синтаксис, 27
compile-file-pathname, переменная, 410
compile-file-truename, переменная, 410
compile-print, переменная, 410
compile-verbose, переменная, 410
call-arguments-limit, константа, 410
call-method, макрос, 335
call-next-method, функция, 335
car, функция, 368
case, макрос, 321
catch, специальный оператор, 321
ccase, макрос, 321
cdr, функция, 368
ceiling, функция, 358
cell-error-name, функция, 347
cerror, функция, 347
change-class, обобщенная функция, 336
char, функция, 378
char-code, функция, 366
char-code-limit, константа, 410
char-downcase, функция, 366
char-equal, функция, 366
char-greaterp, функция, 366
char-int, функция, 366
char-lessp, функция, 366
char-name, функция, 366
char-not-equal, функция, 366
char-not-greaterp, функция, 366
char-not-lessp, функция, 366
char-upcase, функция, 366
char/=, функция, 366
char<, функция, 367
char<=, функция, 367
char=, функция, 366
char>, функция, 367
char>=, функция, 367
character, функция, 366
characterp, функция, 366
check-type, макрос, 347
cis, функция, 358
class-name, обобщенная функция, 336
class-of, функция, 336
clear-input, функция, 389
clear-output, функция, 389
close, функция, 389
clrhash, функция, 384
code-char, функция, 367
coerce, функция, 320
compile, функция, 316
compile-file, функция, 405
compile-file-pathname, функция, 405
compiled-function-p, функция, 321
compiler-macro-function, функция, 317
complement, функция, 321
complex, функция, 358
complexp, функция, 358

compute-applicable-methods, обобщенная функция, 336
compute-restarts, функция, 347
concatenate, функция, 380
concatenated-stream-streams, функция, 390
cond, макрос, 322
conjugate, функция, 358
cons, функция, 368
consp, функция, 369
constantly, функция, 322
constantp, функция, 317
continue, функция, 347
copy-alist, функция, 369
copy-list, функция, 369
copy-pprint-dispatch, функция, 395
copy-readtable, функция, 403
copy-seq, функция, 380
copy-structure, функция, 344
copy-symbol, функция, 351
copy-tree, функция, 369
cos, функция, 358
cosh, функция, 358
count, функция, 380
count-if, функция, 380
count-if-not, функция, 380
ctypecase, макрос, 322
c*x*r, функция, 56, 368

D

debug-io, переменная, 410
debugger-hook, переменная, 410
default-pathname-defaults, переменная, 410
decf, макрос, 358
declaim, макрос, 317
declare, 317
decode-float, функция, 358
decode-universal-time, функция, 406
defclass, макрос, 336
defconstant, макрос, 322
defgeneric, макрос, 337
define-compiler-macro, макрос, 317
define-condition, макрос, 347
define-method-combination, макрос, 338, 339
define-modify-macro, макрос, 322
define-setf-expander, макрос, 322
define-symbol-macro, макрос, 318
defmacro, макрос, 318
defmethod, макрос, 340
defpackage, макрос, 352
defparameter, макрос, 322
defsetf, макрос, 323
defstruct, макрос, 344
deftype, макрос, 320
defun, макрос, 323
defvar, макрос, 323
delete, функция, 382
delete-duplicates, функция, 382
delete-file, функция, 388
delete-if, функция, 383
delete-if-not, функция, 383
delete-package, функция, 353
denominator, функция, 358
deposit-field, функция, 359
describe, функция, 407
describe-object, обобщенная функция, 407
destructing-bind, макрос, 323
digit-char, функция, 367
digit-char-p, функция, 367
directory, функция, 388
directory-namestring, функция, 386
disassemble, функция, 407
do*, макрос, 330
do, макрос, 330
do-all-symbols, макрос, 353
do-external-symbols, макрос, 353
do-symbols, макрос, 353
documentation, обобщенная функция, 407
dolist, макрос, 330
dotimes, макрос, 331
double-float-epsilon, константа, 410
double-float-negative-epsilon, константа, 411
dpb, функция, 359
dribble, функция, 407
dynamic-extent, декларация, 317

E

error-output, переменная, 411
ecase, макрос, 323
echo-stream-input-stream, функция, 390
echo-stream-output-stream, функция, 390
ed, функция, 407
elt, функция, 380
Emacs, 21, 35
encode-universal-time, функция, 407
endp, функция, 369
enough-namestring, функция, 386
ensure-directories-exist, функция, 389
ensure-generic-function, функция, 340
eq, функция, 323

eql, функция, 323
equal, функция, 324
equalp, функция, 324
error, функция, 347
etypecase, макрос, 324
eval, функция, 318
eval-when, специальный оператор, 318
evenp, функция, 359
every, функция, 324
exp, функция, 359
export, функция, 354
expt, функция, 359

F

features, переменная, 411
fboundp, функция, 324
fceiling, функция, 359
fdefinition, функция, 324
ffloor, функция, 359
file-author, функция, 389
file-error-pathname, функция, 389
file-length, функция, 390
file-namestring, функция, 386
file-position, функция, 390
file-string-length, функция, 390
file-write-date, функция, 389
fill, функция, 381
fill-pointer, функция, 376
find, функция, 381
find-all-symbols, функция, 354
find-class, функция, 341
find-if, функция, 381
find-if-not, функция, 381
find-method, обобщенная функция, 341
find-package, функция, 354
find-restart, функция, 347
find-symbol, функция, 354
finish-output, функция, 390
first, функция, 369
flet, специальный оператор, 324
float, функция, 359
float-digits, функция, 359
float-precision, функция, 359
float-radix, функция, 359
float-sign, функция, 359
floatp, функция, 359
floor, функция, 360
fmakunbound, функция, 325
force-output, функция, 390
format, функция, 396
formatter, макрос, 400
fresh-line, функция, 390
fround, функция, 360
ftruncate, функция, 360
ftype, декларация, 317
funcall, функция, 325
function, специальный оператор, 325
function-keywords, обобщенная функция, 341
function-lambda-expression, функция, 325
functionp, функция, 325

G

gensym-counter, переменная, 411
gcd, функция, 360
gensym, функция, 351
gentemp, функция, 351
get, функция, 351
get-decoded-time, функция, 407
get-dispatch-macro-character, функция, 403
get-internal-real-time, функция, 407
get-internal-run-time, функция, 407
get-macro-character, функция, 404
get-output-stream-string, функция, 390
get-properties, функция, 369
get-setf-expansion, функция, 325
get-universal-time, функция, 407
getf, функция, 369
gethash, функция, 384
go, специальный оператор, 325
graphic-char-p, функция, 367

H

handler-bind, макрос, 348
handler-case, макрос, 348
hash-table-count, функция, 385
hash-table-p, функция, 385
hash-table-rehash-size, функция, 385
hash-table-rehash-threshold, функция, 385
hash-table-size, функция, 385
hash-table-test, функция, 385
host-namestring, функция, 386

I

identity, функция, 326
if, специальный оператор, 326
ignorable, декларация, 317
ignore, декларация, 317
ignore-errors, макрос, 348
imagpart, функция, 360
import, функция, 354
in-package, макрос, 354
incf, макрос, 360

initialize-instance, обобщенная функция, 341
inline, декларация, 317
input-stream-p, функция, 390
inspect, функция, 408
integer-decode-float, функция, 360
integer-length, функция, 360
integerp, функция, 360
interactive-stream-p, функция, 391
Interleaf, 21
intern, функция, 354
internal-time-units-per-second, константа, 411
intersection, функция, 369
invalid-method-error, функция, 348
invoke-debugger, функция, 348
invoke-restart, функция, 348
invoke-restart-interactively, функция, 348
isqrt, функция, 360

K

keywordp, функция, 351

L

load-pathname, переменная, 412
load-print, переменная, 412
load-truename, переменная, 412
load-verbose, переменная, 412
labels, специальный оператор, 325
lambda, макрос, 318
lambda-list-keywords, константа, 411
lambda-parameters-limit, константа, 411
last, функция, 369
lcm, функция, 360
ldb, функция, 360
ldb-test, функция, 360
ldiff, функция, 369
least-negative-double-float, константа, 411
least-negative-long-float, константа, 411
least-negative-normalized-double-float, константа, 411
least-negative-normalized-long-float, константа, 411
least-negative-normalized-short-float, константа, 411
least-negative-normalized-single-float, константа, 411
least-negative-short-float, константа, 411
least-negative-single-float, константа, 411
least-positive-double-float, константа, 412
least-positive-long-float, константа, 412
least-positive-normalized-double-float, константа, 412
least-positive-normalized-long-float, константа, 412
least-positive-normalized-short-float, константа, 412
least-positive-normalized-single-float, константа, 412
least-positive-short-float, константа, 412
least-positive-single-float, константа, 412
length, функция, 381
let*, специальный оператор, 326
let, специальный оператор, 326
lisp-implementation-type, функция, 408
lisp-implementation-version, функция, 408
list*, функция, 370
list, функция, 369
list-all-packages, функция, 354
list-length, функция, 370
listen, функция, 391
listp, функция, 370
load, функция, 406
load-logical-pathname-translations, функция, 386
load-time-value, специальный оператор, 318
locally, специальный оператор, 318
log, функция, 361
logand, функция, 361
logandc1, функция, 361
logandc2, функция, 361
logbitp, функция, 361
logcount, функция, 361
logeqv, функция, 361
logical-pathname, функция, 386
logical-pathname-translations, функция, 386
logior, функция, 361
lognand, функция, 361
lognor, функция, 361
lognot, функция, 361
logorc1, функция, 362
logorc2, функция, 362
logtest, функция, 362
logxor, функция, 362
long-float-epsilon, константа, 410
long-float-negative-epsilon, константа, 411
long-site-name, функция, 408
loop, макрос, 331

loop-finish, макрос, 335
lower-case-p, функция, 367

M

macroexpand-hook, переменная, 412
modules, переменная, 412
machine-instance, функция, 408
machine-type, функция, 408
machine-version, функция, 408
macro-function, функция, 319
macroexpand, функция, 319
macroexpand-1, функция, 319
macrolet, специальный оператор, 326
make-array, функция, 376
make-broadcast-stream, функция, 391
make-concatenated-stream, функция, 391
make-condition, функция, 349
make-dispatch-macro-character, функция, 404
make-echo-stream, функция, 391
make-hash-table, функция, 385
make-instance, обобщенная функция, 341
make-instance-obsolete, обобщенная функция, 341
make-list, функция, 370
make-load-form, обобщенная функция, 342
make-load-form-saving-slots, функция, 342
make-package, функция, 354
make-pathname, функция, 386
make-random-state, функция, 362
make-sequence, функция, 381
make-string, функция, 378
make-string-input-stream, функция, 391
make-string-output-stream, функция, 391
make-symbol, функция, 351
make-synonym-stream, функция, 391
make-two-way-stream, функция, 391
makunbound, функция, 351
map, функция, 381
map-into, функция, 381
mapc, функция, 370
mapcan, функция, 370
mapcar, функция, 370
mapcon, функция, 370
maphash, функция, 385
mapl, функция, 370
maplist, функция, 370
mask-field, функция, 362
max, функция, 362
member, функция, 371
member-if, функция, 371
member-if-not, функция, 371
merge, функция, 381
merge-pathnames, функция, 387
method-combination-error, функция, 349
method-qualifiers, обобщенная функция, 342
min, функция, 362
minusp, функция, 362
mismatch, функция, 381
mod, функция, 362
most-negative-double-float, константа, 412
most-negative-fixnum, константа, 412
most-negative-long-float, константа, 412
most-negative-short-float, константа, 412
most-negative-single-float, константа, 412
most-positive-double-float, константа, 413
most-positive-fixnum, константа, 413
most-positive-long-float, константа, 413
most-positive-short-float, константа, 413
most-positive-single-float, константа, 413
muffle-warning, функция, 349
multiple-value-bind, макрос, 326
multiple-value-call, специальный оператор, 326
multiple-value-list, макрос, 326
multiple-value-prog1, специальный оператор, 326
multiple-value-setq, макрос, 326
multiple-values-limit, константа, 413

N

name-char, функция, 367
namestring, функция, 387
nbutlast, функция, 368
nconc, функция, 371
next-method-p, функция, 342
nil, константа, 413
nintersection, функция, 369
no-applicable-method, обобщенная функция, 342
no-next-method, обобщенная функция, 342
not, функция, 327
notany, функция, 327
notevery, функция, 327
notinline, декларация, 317
nreconc, функция, 372
nreverse, функция, 383

nset-difference, функция, 372
nset-exclusive-or, функция, 372
nstring-capitalize, функция, 378
nstring-downcase, функция, 378
nstring-upcase, функция, 379
nsublis, функция, 372
nsubst, функция, 373
nsubst-if, функция, 373
nsubst-if-not, функция, 373
nsubstitute, функция, 384
nsubstitute-if, функция, 384
nsubstitute-if-not, функция, 384
nth, функция, 371
nth-value, макрос, 327
nthcdr, функция, 371
null, функция, 371
numberp, функция, 362
numerator, функция, 362
nunion, функция, 373

O

oddp, функция, 362
open, функция, 391
open-stream-p, функция, 392
optimize, декларация, 317
or, макрос, 327
OS/360, 22
output-stream-p, функция, 392

P

package, переменная, 413
print-array, переменная, 413
print-base, переменная, 413
print-case, переменная, 413
print-circle, переменная, 413
print-escape, переменная, 413
print-gensym, переменная, 414
print-length, переменная, 414
print-level, переменная, 414
print-lines, переменная, 414
print-miser-width, переменная, 414
print-pprint-dispatch, переменная, 414
print-pretty, переменная, 414
print-radix, переменная, 414
print-readably, переменная, 414
print-right-margin, переменная, 414
package-error-package, функция, 355
package-name, функция, 355
package-nicknames, функция, 355
package-shadowing-symbols, функция, 355
package-use-list, функция, 355
package-used-by-list, функция, 355
packagep, функция, 355
pairlis, функция, 371
parse-integer, функция, 363
parse-namestring, функция, 387
pathname, функция, 387
pathname-device, функция, 388
pathname-directory, функция, 388
pathname-host, функция, 387
pathname-match-p, функция, 388
pathname-name, функция, 388
pathname-type, функция, 388
pathname-version, функция, 388
pathnamep, функция, 388
peek-char, функция, 392
phase, функция, 363
pi, константа, 413
plusp, функция, 363
pop, макрос, 371
position, функция, 382
position-if, функция, 382
position-if-not, функция, 382
pprint, функция, 400
pprint-dispatch, функция, 400
pprint-exit-if-list-exhausted, функция, 401
pprint-fill, функция, 401
pprint-indent, функция, 401
pprint-linear, функция, 401
pprint-logical-block, макрос, 401
pprint-newline, функция, 402
pprint-pop, макрос, 402
pprint-tab, функция, 402
pprint-tabular, функция, 402
prin1, функция, 403
prin1-to-string, функция, 403
princ, функция, 402
princ-to-string, функция, 402
print, функция, 402
print-not-readable-object, функция, 402
print-object, обобщенная функция, 402
print-unreadable-object, макрос, 403
probe-file, функция, 389
proclaim, функция, 319
prog*, макрос, 327
prog, макрос, 327
prog1, макрос, 327
prog2, макрос, 327
progn, специальный оператор, 327
progv, специальный оператор, 327
provide, функция, 406
psetf, макрос, 328
psetq, макрос, 328
push, макрос, 371
pushnew, макрос, 371

Q

query-io*, переменная, 415
quote, специальный оператор, 320

R

random-state, переменная, 415
read-base, переменная, 415
read-default-float-format, переменная, 415
read-eval, переменная, 415
read-suppress, переменная, 415
readtable, переменная, 415
random, функция, 363
random-state-p, функция, 363
rassoc, функция, 371
rassoc-if, функция, 372
rassoc-if-not, функция, 372
rational, функция, 363
rationalize, функция, 363
rationalp, функция, 363
read, функция, 404
read-byte, функция, 393
read-char, функция, 393
read-char-no-hang, функция, 393
read-delimited-list, функция, 404
read-from-string, функция, 404
read-line, функция, 393
read-preserving-whitespace, функция, 404
read-sequence, функция, 393
readtable-case, функция, 404
readtablep, функция, 404
realp, функция, 363
realpart, функция, 363
reduce, функция, 382
reinitialize-instance, обобщенная функция, 342
rem, функция, 364
remf, макрос, 372
remhash, функция, 385
remove, функция, 382
remove-duplicates, функция, 382
remove-if, функция, 383
remove-if-not, функция, 383
remove-method, обобщенная функция, 342
remprop, функция, 352
rename-file, функция, 389
rename-package, функция, 355
replace, функция, 383
require, функция, 406
rest, функция, 372
restart-bind, макрос, 349
restart-case, макрос, 349
restart-name, функция, 350
return, макрос, 328
return-from, специальный оператор, 328
revappend, функция, 372
reverse, функция, 383
room, функция, 408
rotatef, макрос, 328
round, функция, 364
row-major-aref, функция, 377
rplaca, функция, 372
rplacd, функция, 372

S

standard-input, переменная, 415
standard-output, переменная, 415
sbit, функция, 377
scale-float, функция, 364
schar, функция, 378
search, функция, 383
second, функция, 369
set, функция, 351
set-difference, функция, 372
set-dispatch-macro-character, функция, 404
set-exclusive-or, функция, 372
set-macro-character, функция, 405
set-pprint-dispatch, функция, 403
set-syntax-from-char, функция, 405
setf, макрос, 328
setq, специальный оператор, 328
shadow, функция, 355
shadowing-import, функция, 355
shared-initialize, обобщенная функция, 342
shiftf, макрос, 329
short-float-epsilon, константа, 410
short-float-negative-epsilon, константа, 411
short-site-name, функция, 408
signal, функция, 350
signum, функция, 364
simple-bit-vector-p, функция, 377
simple-condition-format-arguments, функция, 350
simple-condition-format-control, функция, 350
simple-string-p, функция, 378
simple-vector-p, функция, 377
sin, функция, 364
single-float-epsilon, константа, 410
single-float-negative-epsilon, константа, 411

sinh, функция, 364
sleep, функция, 408
slot-boundp, функция, 343
slot-exists-p, функция, 343
slot-makunbound, функция, 343
slot-missing, обобщенная функция, 343
slot-unbound, обобщенная функция, 343
slot-value, функция, 343
software-type, функция, 408
software-version, функция, 408
some, функция, 329
sort, функция, 383
special, декларация, 317
special-operator-p, функция, 319
sqrt, функция, 364
stable-sort, функция, 383
standard-char-p, функция, 367
step, макрос, 408
store-value, функция, 350
stream-element-type, функция, 393
stream-error-stream, функция, 393
stream-external-format, функция, 393
streamp, функция, 393
string, функция, 378
string-capitalize, функция, 378
string-downcase, функция, 378
string-equal, функция, 378
string-greaterp, функция, 379
string-left-trim, функция, 379
string-lessp, функция, 379
string-not-equal, функция, 379
string-not-greaterp, функция, 379
string-not-lessp, функция, 379
string-right-trim, функция, 379
string-trim, функция, 379
string-upcase, функция, 379
string/=, функция, 379
string<, функция, 380
string<=, функция, 380
string=, функция, 379
string>, функция, 380
string>=, функция, 380
stringp, функция, 379
sublis, функция, 372
subseq, функция, 384
subsetp, функция, 372
subst, функция, 373
subst-if, функция, 373
subst-if-not, функция, 373
substitute, функция, 384
substitute-if, функция, 384
substitute-if-not, функция, 384
subtypep, функция, 320
svref, функция, 377
sxhash, функция, 385
symbol-function, функция, 351
symbol-macrolet, специальный оператор, 319
symbol-name, функция, 352
symbol-package, функция, 352
symbol-plist, функция, 352
symbol-value, функция, 352
symbolp, функция, 352
synonym-stream-symbol, функция, 394

T

t, константа, 415
terminal-io, переменная, 415
trace-output, переменная, 415
tagbody, специальный оператор, 329
tailp, функция, 373
tan, функция, 364
tanh, функция, 364
tenth, функция, 369
terpri, функция, 394
the, специальный оператор, 319
throw, специальный оператор, 329
time, макрос, 409
toplevel, 25
trace, макрос, 409
translate-logical-pathname, функция, 388
translate-pathname, функция, 388
tree-equal, функция, 373
truename, функция, 389
truncate, функция, 364
two-way-stream-input-stream, функция, 394
two-way-stream-output-stream, функция, 394
type, декларация, 317
type-error-datum, функция, 320
type-error-expected-type, функция, 320
type-of, функция, 320
typecase, макрос, 329
typep, функция, 320

U

unbound-slot-instance, функция, 343
unexport, функция, 355
unintern, функция, 356
union, функция, 373
unless, макрос, 329
unread-char, функция, 394
untrace, макрос, 409
unuse-package, функция, 356

unwind-protect, специальный оператор, 329
update-instance-for-different-class, обобщенная функция, 344
update-instance-for-redefined-class, обобщенная функция, 344
upgraded-array-element-type, функция, 377
upgraded-complex-part-type, функция, 364
upper-case-p, функция, 367
use-package, функция, 356
use-value, функция, 350
user-homedir-pathname, функция, 409

V

values, функция, 330
values-list, функция, 330
vector, функция, 377
vector-pop, функция, 377
vector-push, функция, 377
vector-push-extend, функция, 378
vectorp, функция, 377
vi, редактор, 35

W

warn, функция, 350
when, макрос, 330
wild-pathname-p, функция, 388
with-accessors, макрос, 343
with-compilation-unit, макрос, 406
with-condition-restarts, макрос, 350
with-hash-table-iterator, макрос, 386
with-input-from-string, макрос, 394
with-open-file, макрос, 394
with-open-stream, макрос, 394
with-output-to-string, макрос, 395
with-package-iterator, макрос, 356
with-simple-restart, макрос, 350
with-slots, макрос, 343
with-standard-io-syntax, макрос, 405
write, функция, 403
write-byte, функция, 395
write-char, функция, 395
write-line, функция, 395
write-sequence, функция, 395
write-string, функция, 395
write-to-string, функция, 403

Y

y-or-n-p, функция, 395
yes-or-no-p, функция, 395

Z

zerop, функция, 364

А

алгоритм
  бинарный поиск, 75
  быстрая сортировка, 174
  кодирование повторов, 53
  логический вывод, 253
  обратная цепочка логического вывода, 254
  поиск в ширину, 68
  поиск кратчайшего пути, 67
  сопоставление с образцом, 254
атом, 26

Б

бинарное дерево, 57
Брукс, Фредерик, 22
буфер, кольцевой, 137

В

встроенный язык, 274
выражение, 25
  вложенное, 26
  правило вычисления, 27
  самовычисляемое, 25
  цитирование, 27

Г

генерация случайного текста, 149

Д

дерево
  бинарное (двоичное), 58
  двоичное поиска, 85
  сбалансированное, 85

З

замыкание, 119
значение
  множественные, 35, 103
  присваивание, 38
  указатель на значение, 51

И

инкапсуляция, 198
инфиксная нотация, 26
истинность, 30
итерация, 40

К

кавычка, обратная, 173
класс, 187
    родительский, 191
    специфичность, 192
Кнут, Дональд, 16, 220
код
    выравнивание, 35
    парные скобки, 35
    повторное использование, 118
    спагетти, 428
    чтение, 34
кодогенерация, 29, 84, 170
комбинация методов, 197
    операторная, 197
комментирование, 59
константа, 38
контекст
    лексический, 97, 118
куча, 422

Л

Лисп
    выразительная сила, 19
    для численных расчетов, 154
    единообразие, 27
    интерактивность, 25, 33, 117, 223
    код как списки, 29
    происхождение, 19
    символьные вычисления, 253
    типы данных, 28
    функциональная парадигма, 39, 111, 116, 258, 279
ложь, 31
лямбда-выражение, 43

М

Маккарти, Джон, 19, 435
макрознаки, 241
макросимвол, 141
макрос чтения, 141
    диспетчеризуемый, 141
массив
    специализированный, 227
математическая индукция, 59
метод, 186
    вспомогательный, 195
    квалификатор, 195
    первичный, 195
    применимый, 193
    специфичность, 194

Н

наследование, 186, 275
    множественное, 276

О

оператор
    деструктивный, 206
    логический, 32, 259
    специальный, 31, 171
    условный, 31, 99
отладчик, 105
очередь, 206

П

палиндром, 62
память
    автоматическое управление, 69
    висячий указатель, 70
    выделение, 69, 229
    куча, 69
    сборка мусора, 69
    утечка, 70
параметр
    необязательный, 114
    обязательный, 114
    остаточный, 113
    по ключу, 60, 114
переменная, 32
    глобальная, 37, 123
    локальная, 37, 123
    свободная, 119
    специальная, 123
печатные знаки, 82
побочный эффект, 36, 39
предшествование, 190
префиксная нотация, 26, 279
программирование
    прототипирование, 23
    сверху-вниз, 115
    снизу-вверх, 21, 115, 118, 263
    спецификация, 22
    функциональное, 39
профилировщик, 221
пул, 232

Р

рекурсия
    базовый случай, 59, 293
    двойная, 58, 59
    понимание, 34, 58, 126
    хвостовая, 127, 222

С

символ, 29
импортированный, 245
интернированный, 147
как переменная, 32
как слово, 149
ключевое слово, 148
неинтернированный, 176
экспортируемый, 148, 244
слот, 187
список, 29, 48
ассоциативный, 66
блочное представление, 49
вложенный, 50
двусвязный, 212
как код, 170
как множество, 60
как очередь, 68
как последовательность, 61
как стопка, 63
правильный, 65
предшествования, 190, 277
пустой, 29
свойств, 145
структура верхнего уровня, 203
устройство, 30
циклический, 215
по голове и хвосту, 216
стек, 232
считыватель, 36

Т

тип, 44
атомарный, 239
встроенный, 44
иерархия, 44, 239
как множество, 239
конечный, 240
целочисленный, 154
типизация
динамическая, 225
сильная, 224
точечная пара, 65

Ф

функция
глобальная, 111
деструктивная, 62, 70, 208
значение, 32
имя, 32
как агрегат, 34
как значение, 118
как процесс, 34
комбинация функций, 118
локальная, 112
определение, 32
отображающая, 56
параметры, 32
рекурсивная, 33
тело, 32
утилита, 115

Э

экземпляр, 187

Я

ячейка, 48

Издательство "СИМВОЛ-ПЛЮС"

Основано в 1995 году

О нас

Наша специализация – книги компьютерной и деловой тематики. Наши издания – плод сотрудничества известных зарубежных и отечественных авторов, высококлассных переводчиков и компетентных научных редакторов. Среди наших деловых партнеров издательства: O'Reilly, Pearson Education, NewRiders, Addison Wesley, Wiley, McGraw-Hill, No Starch Press, Packt, Dorset House, Apress и другие.

Где купить

Наши книги вы можете купить во всех крупных книжных магазинах России, Украины, Белоруссии и других стран СНГ. Однако по минимальным ценам и оптом они продаются:

Санкт-Петербург:
главный офис издательства –
В.О. 16 линия, д. 7 (м. Василеостровская),
тел. (812) 380-5007

Москва:
московский филиал издательства –
ул. 2-я Магистральная, д. 14В
(м. Полежаевская/Беговая),
тел. (495) 638-5305

Заказ книг

через Интернет http://www.symbol.ru

Бесплатный каталог книг высылается по запросу.

Приглашаем к сотрудничеству

www.symbol.ru

Мы приглашаем к сотрудничеству умных и талантливых авторов, переводчиков и редакторов. За более подробной информацией обращайтесь, пожалуйста, на сайт издательства www.symbol.ru.

Также на нашем сайте вы можете высказать свое мнение и замечания о наших книгах. Ждем ваших писем!